Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.31, NO.2, Apr. 2021 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2021.31.2.263

정적 오염 분석을 활용한 타입스크립트 코드의 보안 취약점 탐지*

문 태 근,[1†] 김 형 식[2‡]
[1,2]성균관대학교 소프트웨어학과 (대학원생, 교수)

Detecting Security Vulnerabilities in TypeScript Code with Static Taint Analysis*

Taegeun Moon,[1†] Hyoungshick Kim[2‡]
[1,2]Department of Computer Science and Engineering, Sungkyunkwan University
(Graduate student, Professor)

요 약

자바스크립트로 작성된 웹 어플리케이션에서 Cross-Site Scripting (XSS), SQL Injection과 같은 검증되지 않은 사용자 입력 데이터로 인해 발생하는 취약점을 탐지하기 위해 오염 분석 기법이 널리 사용되고 있다. 이러한 취약점을 탐지하기 위해서는 사용자 입력 데이터에 영향을 받는 변수들을 추적하는 것이 중요하지만, 자바스크립트의 동적인 특성으로 인해 웹 어플리케이션을 실행해 보지 않고 그러한 변수들을 식별하는 것은 매우 어렵다. 때문에, 기존의 오염 분석 도구들은 대상 어플리케이션을 실행하는 오버헤드가 존재하는 동적 오염 분석을 사용하도록 개발되었다. 본 논문에서는 타입스크립트(자바스크립트의 상위집합) 컴파일러를 활용해 얻은 심볼 정보를 기반으로 데이터의 흐름을 정확히 추적하고, 타입스크립트 코드에서 보안 취약점을 발견하는 새로운 정적 오염 분석 기법을 제안하였다. 제안한 기법은 개발자가 검증되지 않은 사용자 입력 데이터를 포함할 수 있는 변수에 표시를 할 수 있도록 하며, 이를 활용해 사용자 입력 값에 영향을 받는 변수와 데이터를 추적한다. 제안한 기법은 TypeScript 컴파일러에 원활히 통합될 수 있기 때문에, 별도의 도구로 작동하는 기존 분석 도구와 달리 개발자가 개발 과정에서 취약점을 발견할 수 있게 한다. 제안한 기법의 유효성을 확인하기 위해 프로토타입을 구현하였으며, 취약점이 보고된 8개의 웹 어플리케이션을 선정하여 분석을 수행하여 성능을 평가한 결과 기존의 취약점을 모두 탐지할 수 있음을 확인하였다.

ABSTRACT

Taint analysis techniques are popularly used to detect web vulnerabilities originating from unverified user input data, such as Cross-Site Scripting (XSS) and SQL Injection, in web applications written in JavaScript. To detect such vulnerabilities, it would be necessary to trace variables affected by user-submitted inputs. However, because of the dynamic nature of JavaScript, it has been a challenging issue to identify those variables without running the web application code. Therefore, most existing taint analysis tools have been developed based on dynamic taint analysis, which requires the overhead of running the target application. In this paper, we propose a novel static taint analysis technique using symbol information obtained from the TypeScript (a superset of JavaScript) compiler to accurately track data flow and detect security

Received(03. 02. 2021), Modified(03. 25. 2021), Accepted(03. 25. 2021)
* 이 논문은 2021년도 정부(과학기술정보통신부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임 (No.2018-0-00532, 고등급(EAL6 이상) 보안 마이크로커널 개발).
† 주저자, taegeun@skku.edu
‡ 교신저자, hyoung@skku.edu(Corresponding author)

vulnerabilities in TypeScript code. Our proposed technique allows developers to annotate variables that can contain unverified user input data, and uses the annotation information to trace variables and data affected by user input data. Since our proposed technique can seamlessly be incorporated into the TypeScript compiler, developers can find vulnerabilities during the development process, unlike existing analysis tools performed as a separate tool. To show the feasibility of the proposed method, we implemented a prototype and evaluated its performance with 8 web applications with known security vulnerabilities. We found that our prototype implementation could detect all known security vulnerabilities correctly.

Keywords: Taint analysis, Static analysis, Software test, TypeScript, JavaScript

I. 서 론

최근 웹 서비스들은 단순히 정보를 제공하는 문서로서의 기능을 벗어나, 웹 어플리케이션 이라고 불릴 정도로 사용자와의 깊은 상호작용을 제공하고 있다. 사용자와의 상호작용을 기반으로 동적인 웹 컨텐츠를 생성하기 위해 사용자 제출 데이터 (또는 사용자 입력 값) 가 널리 사용되는데, 대표적인 예시로 검색어 쿼리, 로그인 폼, SNS 댓글 등이 있다. 하지만 이러한 사용자 입력 값이 개발자가 의도한 형태가 아닌 악의적인 형태로 조작된다면 웹 어플리케이션에서 의도되지 않은 동작을 일으킬 수 있는 가능성이 있다. 예를 들어, 일반적인 데이터(문자열, 숫자)가 와야할 필드에 사용자가 악의적인 SQL 문을 포함시키면 서버에서 해당 입력 값을 기반으로 SQL 문을 구성할 때 사용자가 포함한 악의적인 SQL 문이 삽입되는 SQL Injection[1] 취약점이 발생할 수 있다. 또한, 사용자 입력 값이 표시 되는 웹 페이지에서 사용자 입력 값에 실행 가능한 자바스크립트 코드 조각이 포함되어 있는 경우 브라우저가 이를 렌더링 할 때 공격자가 삽입한 임의의 자바스크립트 코드가 실행되는 Cross-Site Scripting (XSS)[2] 취약점이 발생할 수 있다. 이처럼 웹 어플리케이션은 그 특성상 적절히 검증되지 않은 사용자 입력 값으로 인해 취약점이 발생는 경우가 많은데, 실제로 Injection 취약점과 XSS 취약점은 OWASP에서 선정한 2017년도 웹 어플리케이션 상위 10개 취약점[3]에 각각 1위와 7위에 등재되었다. 따라서 웹 어플리케이션을 개발할 때 이러한 취약점을 예방하는 것이 중요하고 할 수 있다.

웹 어플리케이션에 존재하는 취약점을 탐지하기 위해 여러 가지 기법들이 제안되었지만, 많은 연구들이 PHP나 Java로 작성된 서버단 코드를 분석 대상으로 하고 있다[4-9]. 이는 웹 취약점이 서버단에서 발생할 여지가 더 많으며, 전통적으로 웹 서버 어플리케이션은 PHP나 Java와 같은 언어로 작성되어왔기 때문이다[10]. 하지만 Node.js가 브라우저 밖에서 자바스크립트 사용을 가능하게 해 주며 자바스크립트가 서버단 어플리케이션 개발에도 사용되기 시작했다. 클라이언트단과 서버단에서 모두 자바스크립트를 사용함으로써 얻을 수 있는 코드 공유등과 같은 많은 이점 덕분에, 자바스크립트를 사용한 서버단 어플리케이션은 최근 들어 많은 인기를 얻고 있다[11]. 따라서 자바스크립트로 작성된 웹 어플리케이션의 취약점 탐지도 중요해 졌다고 이야기 할 수 있다.

웹 어플리케이션에서 취약점을 탐지하기 위한 방법은 소스코드만을 통해 분석하는 정적 분석(static analysis)과 실행 결과를 통해 분석하는 동적 분석(dynamic analysis)이 있다. Karim 등[12]은 어플리케이션 소스코드 조작과 추상적인 실행을 통한 동적 분석 방법을 제안하였다. 또한, Wang 등[13]은 클라이언트단 자바스크립트에서 취약점을 탐지하기 위해 DOM API를 재작성한 웹 브라우저를 제작하였으며, 이를 사용한 자바스크립트 코드 실행을 통한 동적 분석 방법 제안하였다. 비슷하게, Wei 등[14]은 수정된 웹 브라우저를 사용해 코드 실행 결과를 기록하여 이를 정적 분석에 결합하는 혼합 분석 방법을 제안하였다. 하지만 동적 언어인 자바스크립트의 특성상 실제 실행 전에는 변수에 들어있는 데이터가 어떤 타입인지 소스코드만으로는 알아낼 수 있는 정보가 한정적이기 때문에, 선행 연구들은 이를 극복하기 위해 동적 분석을 포함하는 경우가 많아 분석에 실행 오버헤드가 존재한다는 단점이 있으며, 개발 프로세스와 별개로 취약점 검사를 진행해야 하기 때문에 개발자가 능동적으로 보안 분석 과정에 관여하기 어렵다.

본 연구에서는 자바스크립트 코드에서 정적 분석의 어려움를 극복하고, 개발자가 개발 단계에서 보안 분석기와 상호작용하며 취약점을 탐지할 수 있게 하기 위하여 타입스크립트 컴파일러 API를 활용한 정적 오염 분석 기법을 제안하였다. 타입스크립트[16]는 자바스크립트의 상위 집합 언어로, 자바스크립트

의 기본 문법에 타입 표기 문법과 최신 자바스크립트 명세가 추가된 것이며, 순수한 자바스크립트 코드도 타입스크립트 코드라 할 수 있다. 타입스크립트는 강력한 타입 추론 기능을 통해 변수가 의미적으로 어떠한 대상을 나타내는지인 심볼(symbol) 정보(섹션 2.2.1)와 그 데이터 타입 정보를 알아낼 수 있다. 제안한 기법은 이를 활용하여 기존 자바스크립트 코드만으로는 알아낼 수 없었던 심볼 정보와 이들 사이의 데이터 흐름을 분석하여(섹션 4.2) 검증되지 않은 사용자 입력값이 사용되는 지점을 탐지하여 취약점을 개발 단계에서 발견한다. 이를 위해 제안한 기법은 심볼이 기존에 가지는 정보를 확장하여, 심볼의 신뢰성(trust level)과 다른 심볼로의 데이터 흐름 정보를 추가적으로 저장한다. 이때 심볼의 신뢰성은 검증된 값이 들어있는 신뢰(Trust)와 검증되지 않은 값이 들어있는 비신뢰(Untrust)로 나누어지며, Untrust 심볼로부터 Trust 심볼로 데이터가 흘러갈 때 취약점이 발생할 수 있는 지점으로 탐지하게 된다(섹션 4.2.2). 또한 소스코드 상에서 심볼이 선언되는 지점(섹션 4.1)과 사용되는 지점(섹션 4.2.3)에 심볼의 신뢰성을 표기할 수 있게 하여 개발자가 분석기와 상호작용 할 수 있게 하였다. 본 연구가 기여한 바는 다음과 같다.

- ◉ 타입스크립트 컴파일러 API를 통해 얻은 심볼 정보와 타입 정보를 통해 자바스크립트의 정적 분석에 대한 어려움을 극복할 수 있음을 보였다.
- ◉ 관련 연구에서 취약점이 발견 된 6개의 서버단 어플리케이션과 추가로 2개의 클라이언트단 어플리케이션을 대상으로 분석을 수행하여, 제안한 기법이 웹 취약점을 탐지할 수 있음을 확인하였다.
- ◉ 이미 사용되고 있는 타입스크립트 컴파일러를 활용하여, 사용자 입력 값이 유입되는 지점과 검증되는 지점을 코드 내에 직접 명시할 수 있게 하여 개발 과정에서 개발자가 보안 분석에 능동적으로 개입하며 취약점을 탐지할 수 있는 기법을 제안하였다.

본 논문의 구성은 다음과 같다. 2장에서 배경 지식으로 오염 분석과 타입스크립트에 대해 설명하고, 3장에서는 선행 연구를 설명하고 4장에서는 제안한 기법과 작동 원리에 대해 설명한다. 5장에서는 제안한 기법의 유효성에 대한 평가를, 6장에서는 논의 사항을 설명하고 마지막으로 7장에서는 결론과 향후 계획을 제시한다.

II. 배경 지식

2.1 오염 분석(taint analysis)

오염 분석[15]은 민감한 데이터(사용자 입력 값)가 프로그램 내에서 어떻게 흘러가는지 데이터 흐름을 분석하는 방법으로, 보안 취약점을 탐지하는데 사용되는 기법중 하나이다. 구체적으로, 외부의 입력값들을 읽어오는 코드를 오염(taint)의 근원(source)으로 설정하여, 이러한 입력 값들로 인해 취약점이 발생할 수 있는 잠재적인 구멍(sink)들로 오염된 데이터가 전파되는 것을 추적한다. 오염 분석 방법은 다시 정적 오염 분석(static taint analysis), 동적 오염 분석(dynamic taint analysis), 그리고 둘을 함께 사용하는 혼합 분석(hybrid analysis)으로 분류된다. 정적 오염 분석은 프로그램의 실행 없이 소스코드만을 가지고 분석을 수행하는 방법으로 정확성이 떨어진다는 단점이 있으며, 동적 오염 분석은 프로그램의 실행 결과를 통해 분석하는 방법으로 정확성이 높은 대신 실행 오버헤드와 구현이 더 복잡하다는 단점이 존재한다.

2.2 타입스크립트(TypeScript)

타입스크립트[16]는 Microsoft사에서 개발한 프로그래밍 언어로, 자바스크립트를 기반으로 타입 표기 등 추가적인 문법을 더한 상위 집합이며 컴파일 시 일반적인 자바스크립트 코드로 변환된다. 타입스크립트 컴파일러는 강력한 타입 엔진을 통해 변수가 의미적으로 어떠한 대상을 나타내는지를 의미하는 심볼 정보(섹션 2.2.1)와 그 타입을 추론할 수 있으며, 이를 활용해 코드가 문법(syntax)적으로 올바른지 뿐 만 아니라, 의미(semantics)적으로 올바른지도 검사한다.

예를 들어, Fig.1은 두 객체의 name 프로퍼티의 값이 동일한지 비교하는 자바스크립트 코드로, 5번째 줄에서 두 번째 매개변수에 naem으로 오타가 포함되어있다. 자바스크립트에는 변수의 타입에 대한 정보가 없기 때문에, 해당 코드는 실행을 해 보아야 오류가 발생하는 것을 알 수 있다. 반면 Fig.2는 해당 코드를 타입스크립트로 작성한 것으로, equals함수에 선언된 파라미터에 타입 정보를 명시하고 있다.

```
1  function equals(o1, o2) {
2      return o1.name === o2.name;
3  }
4
5  equals({name: "alice"}, {naem: "bob"});
```

Fig. 1. JavaScript Code Example.

```
1  type Human = {
2      name: string;
3  }
4
5  function equals(o1: Human, o2: Human) {
6      return o1.name === o2.name;
7  }
8
9  equals({name: "alice"}, {naem: "bob"});
```

Fig. 2. TypeScript Code Example.

이를 통해 6번째 줄에서 o1.name과 o2.name이 Human 타입의 동일한 필드를 나타내는 심볼이란 사실과, 9번째 줄에서 잘못된 타입의 매개변수가 전달되었음을 컴파일 시점에 알 수 있다.

타입 시스템은 대규모 프로젝트를 유지하고 버그를 조기에 발견하는데 큰 도움이 되기 때문에[17] 많은 기업들이 자바스크립트를 대신하여 타입스크립트를 사용하고 있다. 또한, 웹 개발자를 대상으로 한 State of JS 2020 설문조사에 따르면 99.7%가 타입스크립트에 대해 알고 있으며, 87.7%가 타입스크립트에 관심이 있거나 다시 사용할 계획이 있다고 할 정도로 높은 인기를 가지고 있다[18].

2.2.1 타입스크립트 컴파일러 구조

타입스크립트 컴파일러는 소스코드의 문법적 분석뿐만 아니라 의미적 분석도 수행한다. 아래는 타입스크립트 컴파일러의 5가지의 주요 구성 요소와 각각의 기능을 간략히 설명한 것이다[19].

◉ **Scanner & Parser**: Scanner는 소스코드로부터 개별 문법 요소인 토큰(token)을 식별한다. 예를들어 Fig. 4에서 초록색으로 표현된 노드들이 개별적인 토큰에 해당한다. Parser는 토큰들의 타입과 연관 관계를 트리 형태로 표현한 추상 구문 트리(Abstract Syntax Tree, AST)를 생성한다. AST의 말단 노드는 소스코드에서의 각 토큰을 나타내며, 중간 노드는 자식

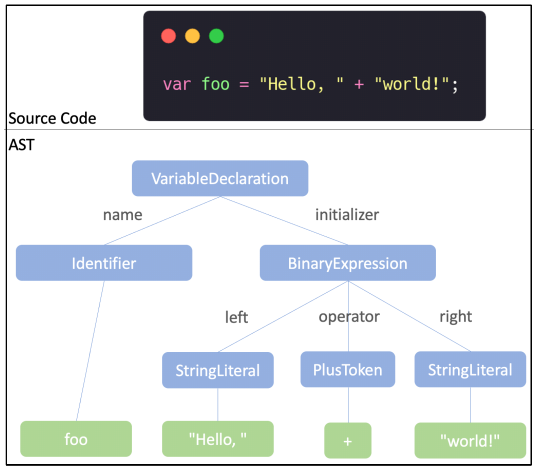

Fig. 3. Example of an Abstract Syntax Tree.

노드들을 묶었을 때 어떤 표현을 의미하는지를 나타낸다. 컴파일러는 AST의 각 노드를 방문하며 주변 노드들과의 관계 정보를 통해 프로그램을 분석한다. Fig.3은 자바스크립트 소스코드와 이를 AST로 나타낸 예시이다.

◉ **Binder**: Binder는 변수, 함수, 타입 등을 나타내는 AST 노드를 그에 상응하는 심볼에 연결한다. 심볼은 프로그램 내에서 같은 의미를 나타내는 요소들을 나타낸다. 예를 들어 Fig.4에서 prop은 총 세 번 나타나지만, ①과 ③은 동일한 심볼에 연결되며, ②는 개별적인 심볼에 연결된다.

```
1  const obj = {
2      prop: "foo" // ①
3  }
4
5  const prop // ②
6      = obj.prop; // ③
```

Fig. 4. Example of Symbols. Same color indicates same symbols.

◉ **Checker**: Checker는 타입 추론과 검증을 수행하는 타입스크립트 컴파일러의 핵심 요소이며, 문법적 내용만으로는 알 수 없는 의미적 내용을 분석하며 내부적으로 Binder를 사용한다. 타입스크립트 컴파일러 API는 Checker를 노출하여 이를 통해 소스코드에서 심볼 정보를 얻을 수 있게 한다.

◉ **Emitter**: Emitter는 분석한 타입스크립트 코드를 상응하는 자바스크립트 코드로 변환한다.

III. 관련 연구

자바스크립트로 작성된 웹 어플리케이션에서 취약점을 탐지하기 위해 다양한 연구가 수행되었다. 특히 웹 어플리케이션의 취약점은 사용자 입력 값에서 기인하는 경우가 많기 때문에, 외부에서 온 데이터의 흐름을 기반으로 코드를 분석하는 오염 분석[15] 기법이 주로 사용되었다.

Karim 등[12]은 소스코드 조작과 플랫폼에 독립적인 추상 실행 환경을 통한 동적 오염 분석 시스템을 제안하였으며, 조작하지 않은 소스코드 대비 최대 38.42배의 실행 오버헤드가 측정되었다. Wang 등[13]은 DOM API를 재작성한 브라우저를 통해 코드를 실행함으로써 DOM 기반 XSS를 탐지하는 동적 오염 분석을 제안하였다. Wei 등[14]은 수정된 웹 브라우저를 사용해 코드 실행 결과를 기록하는 동적 오염 분석과 이를 정적 분석에 결합하는 혼합 분석 방법을 제안하였다. Stock 등[20]은 웹 브라우저 엔진을 수정해 동적 오염 분석을 구현하여 클라이언트단 자바스크립트에서 발생할 수 있는 XSS를 탐지하는 방법을 제안하였다. Lekies 등[21]은 오염 정보를 인지하도록 수정된 자바스크립트 엔진과 DOM API를 사용하여 DOM 기반 XSS를 탐지하는 동적 오염 분석 방법을 제안하였다.

자바스크립트는 그 동적인 특성으로 인해 코드만으로 실행 결과를 예측하기 어렵다는 한계점이 존재한다. 예를 들어, Fig.1에서 2번째 줄에서 o1.name과 o2.name이 동일한 타입의 동일한 필드를 나타낸다는 것은 실제 실행을 해 보기 전까지는 알 수 없다. 때문에 선행 연구들은 주로 동적 분석을 사용하고 있으며, 소스코드를 조작하거나 분석에 실행 오버헤드가 존재한다는 단점이 존재한다. 이 외에도 여러 상용 정적 분석 도구들[22-24]이 존재하지만, 개발 프로세스와 별개로 분석이 수행되어야 하기 때문에 개발자가 능동적으로 분석 과정에 개입하는데 불편함이 있다(섹션 6.1).

IV. 제안 기법

본 연구에서 제안한 기법의 목적은 타입스크립트 소스코드에서 데이터 흐름 분석을 통해 검증되지 않은 사용자 입력 값으로 인해 발생하는 취약점을 탐지하는 것과, 개발자가 데이터의 검증 여부를 코드상에 표기할 수 있게 하여 보안 분석 과정이 개발 과정에 유연하게 포함될 수 있게 하는 것이다. 이를 달성하기 위해 타입스크립트 컴파일러 API를 사용해 얻은 심볼 정보를 기반으로 데이터의 흐름을 분석하는 정적 오염 분석 기법을 제안하였다. 순수한 자바스크립트 코드도 타입스크립트 코드로 볼 수 있으며, 순수 자바스크립트 코드도 JSDoc[25] 문법을 통해 주석 형태로 타입 정보를 표기할 수 있기 때문에 제안한

Algorithm 1: Static Taint Analysis using TypeScript Compiler API

```
    ▷ Define
1   cp = (TypeScript Compiler)
2   AST = cp.getProgramAST()
    ▷ Apply Initial Trust Level (§4.1)
3   for each (node in AST)
4     if (isDeclaration(node)) {
5       symbol = cp.getSymbolAt(node)
6       applyInitialTrustLevel(symbol, node)
7     }
    ▷ Data Flow Analysis (§4.2)
8   for each (node in AST)
9     if (isDataFlow(node)) {
10      LHS = node.LHS
11      RHS = node.RHS
12      tl = getExplicitTrustLevel(RHS)
13      if (tl == Null) {
14        srcSymbols = flowsFrom(RHS)
15        for each (symbol in srcSymbols)
16          connectDataFlow(LHS, symbol)
17      } else if (tl == Untrust) {
18        propagateUntrust(target, Null)
19      }
20    }
    ▷ Connect Data Flow between Symbols (§4.2.1)
21  function connectDataFlow(target, source)
22    source.flowsTo.add(target)
23    if (source.trustLevel == Untrust) {
24      propagateUntrust(target, source)
25    }
```

Fig. 5. Algorithm 1: Static Taint Analysis using TypeScript Compiler API.

기법은 자바스크립트 코드를 분석하는데도 사용될 수 있다. Fig.5는 제안한 기법의 알고리즘을 나타낸 의사코드로, 크게 초기 신뢰성 설정(섹션 4.1), 데이터 흐름 분석(섹션 4.2), 그리고 데이터 흐름 연결(섹션 4.2.1)과 비신뢰 전파(섹션 4.2.2)로 구성된다.

4.1 심볼의 초기 신뢰성

제안한 기법은 심볼이 기존에 가지는 정보들을 확장하여 검증되지 않은 사용자 입력값이 들어있을 수 있는지 여부인 신뢰성 정보(trust level)를 추가적으로 저장한다. 심볼의 신뢰성은 신뢰(Trust), 비신뢰(Untrust), 그리고 정해지지 않음(Null)의 세 가지 상태가 있다. 심볼이 Trust인 경우 검증된 데이터만이 들어갈 수 있으며, Untrust인 경우 검증되지 않아 잠재적으로 취약점을 일으킬 수 있는 데이터가 들어있을 수 있으며, Null인 경우 추후 데이터 흐름 분석과 비신뢰 전파를 통해 심볼의 신뢰성 상태가 업데이트 된다(섹션 4.2.2).

일반적으로 오염 분석 시스템에서는 검증되지 않은 데이터(사용자 입력 값)가 어디서 유래해 오는지(source)와 어디서 사용될 때 취약점이 발생하는지(sink)를 정의한 명세를 바탕으로 검증되지 않은 데이터인 오염의 흐름을 분석한다. 이때 Untrust는 source로, Trust는 sink로 생각할 수 있다. 즉, source에서 유래한 데이터는 신뢰할 수 없으며(Untrust), sink로는 신뢰하는 데이터(Trust)만 들어갈 수 있다. 제안한 기법에서는 이를 위해 소스코드에서 심볼이 생성되는 지점(변수, 함수 선언 등)에 명시적으로 심볼의 신뢰성을 표기할 수 있게 하였으며, 분석이 시작될 때 심볼 선언에 포함된 명시적 신뢰성 정보를 불러온다(Fig.5 3~7번째 줄). 다음은 심볼이 선언되는 지점으로 고려한 AST 노드를 나열한 것이다(Fig.5 4번째 줄). 이때, 함수를 나타내는 심볼의 신뢰성은 함수의 리턴값의 신뢰성을 의미한다.

- **VariableDeclaration**: 변수 선언
- **ParameterDeclaration**: 함수 선언에 포함된 파라미터 선언
- **PropertyDeclaration**: 클래스의 멤버 변수 선언
- **PropertyAssignment**: 객체 리터럴(object literal)의 프로퍼티 선언
- **PropertySignature**: 인터페이스의 프로퍼티 선언
- **FunctionDeclaration**: 함수 선언
- **MethodDeclaration**: 클래스의 멤버 함수 선언
- **MethodSignature**: 인터페이스의 멤버 함수 선언

Fig.6은 변수 선언에 명시적으로 신뢰성을 표기한 예시로, 기존 타입스크립트 컴파일러와의 호환성 유지를 위해 주석을 통해 명시적 신뢰성을 표기하도록 구현하였다.

```
1  var unsafeVar /*@Untrust*/ = "unsafe";
2  var safeVar /*@Trust*/ = "safe";
```

Fig. 6. Example of Explicit Trust Level.

또한, 타입스크립트는 타입 정보가 없는 자바스크립트 라이브러리에 타입 정보만 추가하거나 타입스크립트 라이브러리의 잘못된 타입 정보를 오버라이딩 할 수 있도록 타입 선언 기능[26]을 제공하는데, 제안한 기법은 이를 활용하여 분석 대상 어플리케이션 뿐 만 아니라 외부 라이브러리에 선언된 심볼에 대해서도 신뢰성을 표시할 수 있다. 이러한 기능은 검증되지 않은 데이터 유입과 취약점 발생이 주로 외부 라이브러리를 통해 발생하기 때문에 특히 유용하다. 제안한 기법은 이러한 방식을 통해 기존의 타입스크립트 생태계와 쉽게 통합될 수 있도록 설계되었으며, 7530개의 자바스크립트 라이브러리들에 대한 타입 정보가 정의되어 있는 DefinitelyTyped 프로젝트[27] 등 기존의 방대한 타입 생태계를 활용하여 더 정확한 분석이 가능 할 것이다.

4.2 데이터 흐름 분석

명시적 신뢰성 표기를 통해 심볼의 초기 신뢰성이 결정된 이후, 데이터 흐름 분석을 통해 심볼의 데이터가 어디로 이동하는지 추적하게 된다(Fig.5 8~20번째 줄). 본 연구에서는 다양한 종류의 데이터의 흐름을 $LHS = RHS$ 의 표기법을 사용하여 할당(assignment)으로 일반화 하였다. 이때 LHS는 데이터가 흘러 들어가는 대상 심볼이며, RHS는 데이터가 유래해오는 심볼이다. 구체적으로 고려한 데이터 흐름은 아래와 같다.

◉ **Assignment**: 변수에 값을 할당하는 일반적인 경우이며, *LHS* = *RHS*로 표현할 수 있다.
◉ **Return**: 함수에서 값을 반환하는 경우이며, 이 경우 반환되는 값들이 *RHS*이며, *LHS*는 함수를 나타내는 심볼에 함수의 반환값을 나타내는 특별한 심볼로 저장하게 하여 일반화 하였다.
◉ **Argument Pass**: 함수 호출에서 매개변수는 실제로 전달되는 값이며, 파라미터는 함수 선언에 포함된 변수이다. 이때 전달되는 매개변수는 *RHS*, 함수 선언에 포함된 파라미터(parameter)는 *LHS*로 취급하여 일반화 하였다.

예를 들어, Fig.7에서 5번째 줄의 함수 호출은 1번째 줄에 선언된 파라미터 arg에 매개변수 untrustStr를 할당하는 것으로 취급하였다.

```
1  function identity(arg) {
2      return arg;
3  }
4  const untrustStr /*@Untrust*/ = "MALICIOUS";
5  identity(untrustStr);  // arg = untrustStr
```

Fig. 7. Example of an Argument Pass.

4.2.1 데이터 흐름 연결

컴파일러가 생성한 AST의 각 노드를 방문하며, 데이터의 흐름을 나타내는 노드를 발견하면(Fig.5 9번째 줄) *RHS*가 나타내는 심볼에서 *LHS*로 데이터가 흘러간다는 정보를 추가한다. 이때 *LHS*는 하나의 심볼로 결정되지만 *RHS*는 복잡한 표현식(expression)일 수 있기 때문에 둘 이상의 심볼이 포함되어 있을 수 있다. 이 경우 *RHS*에 포함된 심볼 중 *LHS*로 데이터가 흘러간다고 취급하는 심볼들의 집합 나타내기 위해 *flowsFrom(RHS)*의 표기법을 사용하며(Fig.5 14번째 줄), *flowsFrom(RHS)*에 포함된 심볼들 각각에 *LHS*를 나타내는 심볼로 데이터가 흘러간다는 정보를 추가한다(Fig.5 14~16번째 줄). 이때, AST에서 *RHS*를 나타내는 표현식 노드에 명시적 신뢰성 표기가 있는 경우, 데이터 흐름 연결이 발생하지 않는다(섹션 4.2.3). *RHS*에 올 수 있는 표현식으로 고려한 사항과 예시, 그리고 각각에 해당하는 *flowsFrom(RHS)*는 Table 1.과 같다.

◉ **Identifier**: 하나의 심볼을 나타내므로, 해당 심볼 연결.
◉ **Binary**: *flowsFrom(expr1)*과 *flowsFrom(expr2)*에 속한 모든 심볼과 연결.
◉ **Conditional**: *flowsFrom(expr1)*과 *flowsFrom (expr2)*에 속한 모든 심볼과 연결.
◉ **ObjectLiteral**: 객체의 프로퍼티를 나타내는 모든 심볼과 연결.
◉ **Template**: 치환되는 부분에 들어가는 모든 표현식에 대하여 재귀적으로 연결.
◉ **PropertyAccess**: 해당 프로퍼티를 나타내는

Table 1. Expressions Containing Multiple Symbols.

RHS Expression	RHS Example	Symbols flow to LHS (flowsFrom(RHS))
Identifier	s1	Symbol represented by the identifier “s1”
Binary	expr1 + expr2	flowsFrom(expr1) ∪ flowsFrom(expr2)
Conditional	flag ? expr1 : expr2	flowsFrom(expr1) ∪ flowsFrom(expr2)
ObjectLiteral	{ foo: expr1, bar: expr2 }	flowsFrom(foo) ∪ flowsFrom(bar)
Template	`${expr1} and ${expr2}`	flowsFrom(expr1) ∪ flowsFrom(expr2)
PropertyAccess	foo.bar	Symbol represented by the identifier “bar”
ElementAccess	foo[bar]	Symbols of all properties in “foo”
Parenthesized	(expr1)	flowsFrom(expr1)
Call	someFunc(expr1, expr2)	Symbol of return value of the function if function is internal, flowsFrom(expr1) ∪ flowsFrom(expr2) otherwise

심볼과 연결.

◉ **ElementAccess**: 해당 객체에 선언된 모든 프로퍼티와 연결.

◉ **Parenthesized**: 괄호를 푼 상태의 표현식으로 취급하여 심볼 연결.

◉ **Call**: 호출되는 함수가 내부 함수인 경우 해당 함수의 리턴값을 나타내는 심볼과 연결, 외부 함수인 경우 전달된 매개변수에 포함된 모든 심볼과 연결.

이때 함수 호출의 경우, 함수가 라이브러리에서 선언된 외부 함수인지 아닌지에 따라 다르게 취급하는데, 이는 함수가 분석 대상 어플리케이션 코드에 선언된 경우는 리턴 값을 나타내는 심볼이 데이터 흐름 분석을 통해 정확한 추적이 가능하지만, 라이브러리에 선언된 함수의 구현부는 분석 대상에 들어가지 않기 때문에 리턴값의 신뢰성이 전달한 매개변수에 달려있다 여기기 때문이다.

4.2.2 비신뢰 전파

데이터 흐름 분석을 통해 두 심볼 사이의 데이터 흐름이 연결될 때, 데이터가 유래해 오는 심볼을 source, 흘러 들어가는 대상 심볼을 target이라 칭한다. 이때 source의 신뢰성이 Untrust라면 데이터 흐름 규칙(data flow rule)에 따라 target의 신뢰성을 Untrust로 업데이트 하는 비신뢰 전파가 발생하며(Fig.5 23~24번째 줄), 이 과정에서 데이터 흐름 규칙을 위반하는 경우를 통해 취약점이 발생할 수 있는 지점을 탐지한다. Table 2.는 target의 성질에 따라 비신뢰 전파가 어떻게 작용하는지를 정의한 데이터 흐름 규칙을 나타낸 표이며, Fig.8는 전파 알고리즘을 나타낸 의사 코드이다.

우선 target의 현재 신뢰성이 Trust인 경우, 검증되지 않은 source의 데이터를 검증된 데이터가 들어있어야 하는 target에 할당하는 경우이므로 취약점이 발생할 수 있는 지점으로 탐지하여 보고하며(Fig.8 2~3번째 줄) 데이터 흐름을 허용하지 않는다. 다음으로 target의 신뢰성이 Untrust라면, 이

Table 2. Data flow rule.

Type of target	Trust	Untrust	Null
Internal Symbol	X	-	Propagate
External Symbol	X	-	-

Algorithm 2: Untrust Propagation

```
    ▷ Propagate Untrust (§4.2.2)
1   function propagateUntrust(target, source)
2     if (target.trustLevel == Trust) {
3       reportViolation()
4     } else if (target.trustLevel == Null) {
5       if ( ! target.isExternal ) {
6         target.trustLevel = Untrust
7         for each (flow in target.flowsTo)
8           propagateUntrust(flow, target)
9       }
10    }
```

Fig. 8. Algorithm 2: Untrust Propagation.

미 target으로부터 비신뢰 전파가 수행 되었을 것이므로 전파를 진행하지 않은채 데이터 흐름은 허용된다. 마지막으로, target의 신뢰성이 아직 정해지지 않은 Null 상태인 경우 target이 외부 라이브러리에 선언된 외부 심볼인지 현재 어플리케이션 코드에 선언된 내부 심볼인지에 따라 다르게 처리한다. target이 내부 심볼인 경우, 신뢰성을 Untrust로 업데이트 하고 target을 시작으로 다시 데이터 흐름을 따라 재귀적으로 비신뢰 전파를 수행한다(Fig.8 4~9번째 줄). target이 외부 심볼인 경우 해당 라이브러리 내의 취약점을 찾는 것이 목적이 아니므로 전파를 수행하지 않으며, 신뢰성 여부가 상관없는 경우라 취급하여 데이터 흐름을 허용한다. 이 과정을 통해 Untrust 심볼이 흘러 들어가는 모든 대상 심볼들의 신뢰성 또한 재귀적으로 Untrust로 업데이트 되며, Untrust 심볼이 Trust로 흘러 들어가는 것을 취약점이 발생할 수 있는 지점으로 탐지한다.

4.2.3 표현식의 명시적 신뢰성 표기

앞서 언급하였듯, 제안한 기법은 심볼에 들어있는 데이터가 검증 되었는지 여부를 개발자가 코드상에 명시하는 것을 허용한다. 이러한 기능은 특히 데이터가 악의적인 내용을 포함하는지 검증하는 유틸리티 함수를 작성하여 사용하는 경우 유용한데, 해당 함수의 리턴값을 Trust로 표기하면 자동으로 해당 함수가 사용된 부분은 데이터가 검증되었다고 판단되기 때문이다. 또한 취약점이 아닌 부분을 잘못 탐지하는 오탐이 발생하는 경우, 이를 개발자가 소스코드에서

직접 Trust로 표시하여 분석기에게 알려주는 것이 가능하다.

이를 위해 제안한 기법에서 개발자는 *RHS*의 표현식이 나타내는 데이터가 검증 되었는지 여부를 명시적으로 표시할 수 있게 한다. 이 경우, 데이터 흐름 연결과 비신뢰 전파는 발생하지 않는다. 예를 들어 Fig.9는 XSS 공격을 방지하기 위해 HTML 엘리먼트에 사용자 입력 값을 렌더링 하기 전 HTML 특수문자를 이스케이프 처리하는 sanitizeHTML를 호출하는 코드로, 이를 통해 사용자 입력 값을 검증하게 된다. 이를 분석기에게 알려주기 위해, 5번째 줄의 데이터 흐름에서 *RHS*에 명시적으로 Trust를 표기한 것을 8번째 줄에서 확인할 수 있다. 비슷하게, 개발자는 *RHS*가 검증되지 않은 데이터가 포함될 수 있다고 판단하는 경우 Untrust로 표기할 수 있다. 단, 이 경우 target을 시작으로 다시 비신뢰 전파가 수행된다.

```
1  function display(e: HTMLElement, userInput: string) {
2      e.innerHTML = sanitizeHTML(userInput);
3  }
4  function sanitizeHTML(content: string) {
5      const sanitized = content
6          .replace('<', '&lt;')
7          .replace('>', '&gt;')
8          .replace('&', '&amp;') /*@Trust*/;
9      return sanitized;
10 }
```

Fig. 9. Example of the Explicit Trust Level on RHS.

4.3 분석 예시

이번 섹션에서는 지금까지 설명한 내용을 바탕으로, 실제 코드를 보며 제안한 기법이 어떻게 검증되지 않은 데이터로인해 발생하는 취약점을 탐지하는지 설명한다. Fig.10은 웹 어플리케이션 프레임워크인 express[28] 와 mysql 라이브러리[29]를 사용하는 웹 서버 어플리케이션에서 로그인 요청을 처리하는 부분을 단순화 하여 나타낸 것이다. 해당 코드에는 HTTP 요청에 포함된 사용자 입력 값인 pw를 적절한 검증 없이 16번째 줄의 query 함수에서 쿼리문에 포함시켜, SQL Injection이 발생하는 취약점이 존재한다.

해당 코드에서 1~2번째 줄은 mysql 라이브러리의 타입 선언 파일 일부를, 4~9번째 줄은 express 프레임워크의 타입 선언 파일 일부를 단순화 하여 표시한 것이다. 제안한 기법을 활용하여 정적 오염 분석을 수행하면, 다음과 같은 순서로 데이터의 흐름 규칙을 검사하게 된다.

```
1  // function is declared in "mysql" library
2  declare function query(sql /*@Trust*/: string);
3
4  // interface is declared in "express" library
5  declare interface Request {
6      params: {
7          pw /*@Untrust*/: string;
8      }
9  }
10
11 // Web server application
12 const app = express();
13 app.get('/login', (req: Request, res) => {
14     const password = req.params.pw;
15
16     query("select * from user where pw=" + password);
17 })
```

Fig. 10. Example of the Untrust Propagation.

1. 우선, 섹션 4.1에서 소개한 명시적 신뢰성 표기를 통해, 사용자 입력 값이 들어있는 심볼인 pw(Fig.10 7번째 줄)을 Untrust로 표시하며, 검증된 값들만 매개변수로 전달해야 하는 query 함수의 sql 파라미터(Fig.10 2번째 줄)은 Trust로 표시한다.
2. 다음으로, 14번째 줄에서 req.params.pw가 나타내는 심볼이 7번째 줄에서 선언된 pw인 것을 컴파일러 API를 통해 알아내고, 4.2.1에서 소개한 데이터 흐름 분석을 통해 password가 나타내는 심볼로 흐름이 연결된다.
3. 이때 *RHS*인 req.params.pw가 나타내는 심볼이 Untrust이기 때문에, 4.2.2에서 소개한 비신뢰 전파를 통해 password 또한 Untrust로 신뢰성이 업데이트 된다.
4. 다음으로, 16번째 줄에서 query 함수를 호출 할 때 Argument Pass가 발생하며, 2번째 줄에서 선언된 파라미터 sql이 나타내는 심볼이 *LHS*, 매개변수로 전달된 문자열이 *RHS*가 된다.
5. 이때 *RHS*는 BinaryExpression으로, Table 1.을 통해 password 심볼이 sql 심볼로 데이터 흐름이 연결됨을 알 수 있다.
6. 마지막으로, 해당 데이터 흐름이 연결될 때 source인 password의 심볼이 Untrust 이고 target이 Trust이므로, 데이터 흐름 규칙 위반이 보고되어 취약점을 탐지하게 된다.

V. 평 가

제안한 기법이 실제로 취약점을 탐지할 수 있는지 유효성을 확인하기 위해, 527줄의 타입스크립트 코드로 작성된 프로토타입을 구현하여 실제 취약점이 탐지되었던 웹 어플리케이션을 대상으로 분석을 수행하였다. 실험은 4Core 2.5GHz Intel CPU, 8GB RAM과 Ubuntu 20.04 LTS 운영체제가 설치된 컴퓨터에서 진행되었다.

5.1 실험 대상 선정

5.1.1 오염 명세(taint specification)

오염 분석을 수행하기 위해서는 어디에서 안전하지 않은 데이터(사용자 입력 값)가 유래해 오는지와, 어디에서 사용될 때 취약점이 발생할 수 있는지를 사전에 정의해 놓은 오염 명세가 필요하다. 본 연구에서는 Staicu 등[30]이 수행한 Node.js 자바스크립트 라이브러리에서 오염 명세를 자동으로 추출하는 연구에서 공개한 내용을 기반으로, 8개의 라이브러리에서 28개의 명세를 선정하였다. 또한, 클라이언트단 자바스크립트에서의 취약점을 탐지하기 위해 발견된 취약점을 기반으로 DOM 라이브러리와 jQuery 라이브러리에서 총 5개의 오염 명세를 추가하여 총 10개의 라이브러리에서 20개의 source와 13개의 sink로 총 33개의 오염 명세를 선정하였다. Table 3.은 고려한 라이브러리 목록과 각각의 source와 sink 개수, 사용되어지는 실험 대상 웹 어플리케이션을 나타낸 것 이다.

Table 3. Taint Specification Used in the Experiment.

Library	Source	Sink	Applications
globals.d.ts	1	0	[33-40]
fs	6	3	[34]
http	1	0	[33-38]
child-process-promise	0	4	[33]
qs	1	0	[33-38]
express	9	0	[33-38]
mkdirp	0	1	[34,35]
rimraf	0	2	[34,35]
lib.dom.d.ts	1	1	[39]
jQuery	1	2	[40]
Total	**20**	**13**	-

5.1.2 실험 대상 웹 어플리케이션

실험 대상 웹 어플리케이션으로는 Staicu 등[31]이 수행한 연구를 통해 Injection과 XSS 취약점을 발견한 서버단 자바스크립트 웹 어플리케이션[32]중 6개[33-38]를 선정하였으며, 추가적으로 클라이언트단 어플리케이션에 대한 탐지를 평가하기 위해 XSS 취약점이 보고되었던 2개의 웹 어플리케이션[39,40]

Table 4. Analysis Result on the Web Applications with Known Vulnerability.

App	File Size	Exec Time	Declared Symbols	Untrust Symbols	# of Violation	New Detection	Vulnerability
[33]	2.3kb	1,555ms	32	7	1	0	command-injection
[34]	7.5kb	1,780ms	102	25	3	2	path-injection
[35]	5.0kb	1,534ms	81	6	1	0	path-injection
[36]	1.7kb	1,754ms	35	5	1	0	reflected-xss
[37]	1.8kb	1,526ms	48	13	5	3	reflected-xss
[38]	4.6kb	1,672ms	40	4	2	0	reflected-xss
[39]	9.0kb	1,593ms	94	9	1	0	dom-based-xss
[40]	8.6kb	2,218ms	55	12	2	0	dom-based-xss
Total	-	-	**487**	**81**	**16**	**5**	-

을 선정하였다. 해당 웹 어플리케이션들은 앞서 선정한 라이브러리들을 사용하고 있기 때문에, 제안 기법이 이들로 인한 취약점을 탐지하기 위해 앞서 작성한 오염 명세가 필요했으며, 이를 기반으로 분석을 수행하여 성능을 평가하였다. 분석은 각 웹 어플리케이션에 존재하는 파일 중 취약점이 발견된 소스파일을 대상으로 수행하였으며, 파일의 평균 사이즈와 표준편차는 각각 4308 바이트와 2869 바이트이다.

기존의 취약점 탐지 연구는 대부분 자바스크립트 어플리케이션에 대하여 수행되었기 때문에, 기존 연구에서 취약점을 발견한 웹 어플리케이션들은 대부분이 자바스크립트로 작성되어 있었다. 제안한 기법은 타입스크립트의 타입 추론 기능을 통해 정확한 심볼을 식별하기 때문에, 이를 위해 선정된 웹 어플리케이션들에 적절한 타입 표시를 추가하여 타입스크립트 코드로 변환하였다. 이때 타입 표시 문법은 코드의 동작에 영향을 주지 않았다.

5.2 취약점 탐지 정확성 평가

5.1.2에서 선정한 웹 어플리케이션에 대하여 분석에 걸린 시간(10회 반복 평균), 총 선언된 심볼 수, Untrust로 판단된 심볼 수, 취약점으로 탐지된 횟수, 그리고 실제 취약점이었던 탐지와 새로운 취약점 탐지가 몇 번 발생하였는지 측정하였다. Table 4.는 각 웹 어플리케이션에 대하여 분석을 수행한 결과를 정리한 것이다.

실험 결과 8개 웹 어플리케이션에 선언된 총 487개의 심볼 중 약 17%인 81개의 심볼이 Untrust로 식별되었으며, 총 16개의 데이터 흐름 규칙 위반을 탐지하였다. 탐지된 흐름 규칙 위반을 수작업으로 검증한 결과, 분석을 통해 8개의 웹 어플리케이션 모두에서 기존의 취약점을 탐지할 수 있었으며, [34]와 [37]의 경우 평가 기준으로 사용한 취약점 데이터셋[32]에 포함되지 않았던 새로운 취약점이 추가적으로 각각 2개와 3개 탐지되었음을 확인하였다. 실험에서 오염 명세와 웹 어플리케이션 선정에 수작업이 필요했기 때문에 소수만을 선정할 수 있었으며, 이로 인해 오탐은 확인할 수 없었다. 하지만 오탐이 발생하는 경우에도, 4.2.3에서 언급한 표현식의 명시적 신뢰성 표시를 통해 개발자가 소스코드에 데이터의 안전성을 표기하여 더 이상 오탐으로 탐지되지 않도록 개발 프로세스에서 유연하게 대처할 수 있을 것이다.

VI. 논 의

6.1 기존 정적 분석 도구 대비 특성 비교

Table 5.는 제안한 기법이 가지는 특성을 기존의 정적 분석 도구[22-24]들과 비교한 것이다. 우선, 기존의 도구들은 개발 단계 이후 독립적인 분석 절차를 필요로 하기 때문에 주로 소프트웨어 분석가에 의해 사용된다. 이 경우 취약점이 발견될 때, 분석가가 개발자에게 수정을 요구하는 등 절차가 복잡해진다는 단점이 있다. 반면 제안한 기법은 개발 단계에서 분석을 수행할 수 있기 때문에 개발자에 의해 사용될 수 있으며, 개발자에게 보안 의식을 제고하며 코드 레벨에서 보안 조치를 더 강화할 수 있다. 또한 소프트웨어 결함은 늦게 발견할수록 수정에 드는 비용이 증가하기 때문에[41], 제안한 기법을 통해 취약점을 발견했을 때 개발자가 이를 수정하는데 필요한 비용이 기존 도구에 비해 상대적으로 적다는 장점이 있다. 하지만 취약점 탐지만을 수행하는 제안 기법과 달리, 기존 도구들은 버그 탐지, 코드 품질 검사 등 부가적인 기능도 수행할 수 있다는 장점이 있는 반면, 제안 기법은 취약점 탐지만을 목표로 한다는 한계가 있다.

Table 5. Attributes of the Proposed Technique Compared to the Existing Tools.

Attribute	Existing Tools [22-24]	Proposed Technique
Performed Stage	After the Development Process	During the Development Process
Performed by	Software Analyst	Developer
Cost of Fixing Defects	High	Low
Usage	Vulnerability Detection, Bug Detection, Code Quality	Vulnerability Detection Only

6.2 한계점

제안한 기법은 타입스크립트로 작성된 어플리케이

션을 대상으로 하므로, 자바스크립트로 작성된 어플리케이션을 분석하기 위해서는 오염 정보를 알 수 있는 최소한의 타입 정보를 추가해야 한다. 기존 연구들이 실험 대상으로 사용한 웹 어플리케이션들은 자바스크립트로 작성된 경우가 대부분이기 때문에, 이를 활용하여 제안한 기법을 평가하기 위해 타입 정보를 표기하는 과정이 필요했다. 또한 기존의 오염 명세는 소스코드와 별개의 데이터로 관리되기 때문에 이를 소스코드에 수작업으로 옮기는 과정이 필요했다. 이처럼 실험 데이터 구성에 많은 수작업을 필요로 했기 때문에, 다양한 종류의 웹 어플리케이션에 대한 실험을 하는데 어려움이 있었으며, 다량의 오염 명세를 반영하지 못해, 이로 인해 발생할 수 있는 오탐에 관한 분석이 수행되지 못하였다.

따라서 자바스크립트 코드를 타입스크립트로 변환하는 과정과 오염 명세를 기존의 타입 정보들에 수작업으로 옮기는 과정에 대해서 개발자에 대한 사용성을 개선할 수 있는 방법과, 다량의 오염 명세로 인해 발생하는 오탐에 대한 분석을 추가적으로 연구할 필요가 있다.

6.3 타입 표기 및 오염 명세 변환 자동화 방법론

타입스크립트 컴파일러는 심볼이 선언 될 때의 정보를 기반으로 타입을 추론할 수 있기 때문에 자바스크립트 코드에서도 어느정도 타입 정보를 알아내는 것이 가능하다. 이를 이용하더라도 알아낼 수 없는 타입 정보들은 데이터 흐름 분석과 심볼의 사용 시점 정보를 통해 자바스크립트 코드의 타입 정보를 추론하는 정적 타입 분석기인 Flow[42]와 같은 방법론을 차용하면 더 정확한 타입 추론이 가능할 것이다. 이를 바탕으로, 자바스크립트 코드를 입력 값으로 받아 Emitter 기능을 이용해 추론한 타입 정보를 코드상에 표기만 해 주어도 유효한 타입스크립트 코드를 생성할 수 있으므로, 변환을 자동화 할 수 있을 것이다. 하지만 완벽한 변환을 위해서는 수작업이 요구될 수 있으므로, 자동화 변환시 요구되는 비용은 추후 자동화 도구 개발을 통한 연구 및 분석이 필요하다.

기존의 오염 명세를 소스코드에 표기하는 과정의 경우, 기존의 명세 형식들도 결국 자바스크립트 코드에서의 특정 심볼을 나타내는 것이므로, 기존의 오염 명세 문법을 코드상의 위치로 변환하는 방법을 통해 쉽게 자동화 할 수 있을 것이다.

7. 결론 및 향후 계획

본 논문에서는 타입스크립트 컴파일러 API를 활용한 정적 오염 분석 기법을 통해 타입스크립트로 작성된 어플리케이션에서 검증되지 않고 사용되는 사용자 입력 값으로 인해 발생할 수 있는 취약점을 컴파일 시점에 탐지하는 기법을 제안하였다. 제안한 기법은 타입스크립트 컴파일러 API가 추론한 심볼 정보를 기반으로 기존 자바스크립트 코드의 정적 분석의 어려움을 극복하며, 소스코드에 심볼과 표현식의 신뢰성을 표기할 수 있게 함으로써 개발자가 개발 과정에서 보안 분석 프로세스에 개입하며 취약점을 탐지할 수 있게 하였다. 또한 취약점이 발견되었던 8개의 웹 어플리케이션을 대상으로 분석을 수행한 결과 제안한 기법이 기존의 취약점을 탐지해 낼 수 있음을 확인하였다.

향후 연구에서는 자바스크립트 최신 문법을 반영한 보다 정밀한 데이터 흐름 분석 구현, 대규모 실험 데이터 구성을 통한 정확성 평가와 오탐에 대한 자세한 분석, 그리고 자바스크립트 코드 변환 자동화 방법을 연구할 것이다.

References

[1] SQL Injection | OWASP, https://owasp.org/www-community/attacks/SQL_Injection, accessed on Feb. 28. 2021

[2] Cross Site Scripting (XSS) Software Attack | OWASP Foundation, https://owasp.org/www-community/attacks/xss/, accessed on Feb. 28. 2021

[3] OWASP Top 10 Application Security Risks - 2017, https://owasp.org/www-project-top-ten/2017/Top_10.html, accessed on Feb. 28. 2021

[4] K. Cao, J. He, W. Fan, W. Huang, L. Chen, and Y. Pan, "PHP vulnerability detection based on taint analysis," 6th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO), pp. 436-439, Sept. 2017

[5] R. Jahanshahi and A. Doupé, and Manuel Egele, "You shall not pass:

Mitigating SQL Injection Attacks on Legacy Web Applications," Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, pp. 445-457, Oct. 2020

〔6〕 S.F. Syed, A. Ahmed, G. D'mello, and Z. Ansari, "Removal of Web Application Vulnerabilities using Taint Analyzer and Code Corrector," 2019 International Conference on Nascent Technologies in Engineering (ICNTE), pp. 1-7, Jan. 2019

〔7〕 W. Huang, Y. Dong, and A. Milanova, "Type-Based Taint Analysis for Java Web Applications," Fundamental Approaches to Software Engineering. FASE 2014, pp. 140-154, Apr. 2014

〔8〕 B. Livshits and M. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis," 14th USENIX Security Symposium, Jul. 2005

〔9〕 M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and Flexible Discovery of PHP Application Vulnerabilities," 2017 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 334 - 349, Apr. 2017

〔10〕 Usage statistics of server-side programming languages for websites, https://w3techs.com/technologies/overview/programming_language, accessed on Feb. 28. 2021

〔11〕 J. Strimpel and M. Najim, Building Isomorphic JavaScript Apps, ISBN: 9781491932933, O'Reilly Media, Inc., pp. 3-13, Sept. 2016

〔12〕 R. Karim, F. Tip, A. Sochůrková, and K. Sen, "Platform-Independent Dynamic Taint Analysis for JavaScript," IEEE Transactions on Software Engineering, vol. 46, no. 12, pp. 1364-1379, Dec. 2020

〔13〕 R. Wang, Guangquan Xu, Xianjiao Zeng, X. Li, and Z. Feng, "TT-XSS: A novel taint tracking based dynamic detection framework for DOM Cross-Site Scripting," J. Parallel Distributed Comput, vol. 118, pp. 100-106, Aug. 2018

〔14〕 S. Wei and B. Ryder. "Practical blended taint analysis for JavaScript," Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013), pp. 336-346, Jul. 2013

〔15〕 Static Code Analysis Control | OWASP Foundation, https://owasp.org/www-community/controls/Static_Code_Analysis#taint-analysis, accessed on Feb. 28. 2021

〔16〕 TypeScript: Typed JavaScript at Any Scale, https://www.typescriptlang.org, accessed on Feb. 28. 2021

〔17〕 Z. Gao, C. Bird, and E. Barr, "To Type or Not to Type: QuantifyingDetectable Bugs in JavaScript," 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 758 - 769, May. 2017

〔18〕 State of JS 2020: JavaScript Flavors, https://2020.stateofjs.com/en-US/technologies/javascript-flavors, accessed on Feb. 28. 2021

〔19〕 TypeScript Compiler Internals, https://basarat.gitbook.io/typescript/overview, accessed on Feb. 28. 2021

〔20〕 B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, "Precise Client-side Protection against DOM-based Cross-Site Scripting," 23rd USENIX Security Symposium, pp.655-670, Aug. 2014

〔21〕 S. Lekies, B. Stock, and M. Johns, "25 million flows later: large-scale detection of DOM-based XSS," Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pp. 1193-1204, Nov. 2013

〔22〕 LGTM - Continuous security analysis,

https://lgtm.com, accessed on Feb. 22. 2021
[23] SonarQube - Code Quality and Code Security, https://www.sonarqube.org, accessed on Feb. 22. 2021
[24] DeepSource: Automate code reviews with static analysis, https://deepsource.io, accessed on Feb. 22. 2021
[25] Use JSDoc: Index, https://jsdoc.app, accessed on Feb. 28. 2021
[26] Declaration Files, https://www.typescriptlang.org/docs/handbook/declaration-files/introduction.html, accessed on Feb. 28. 2021
[27] DefinitelyTyped/DefinitelyTyped, https://github.com/DefinitelyTyped/DefinitelyTyped, accessed on Feb. 28. 2021
[28] Express - Node.js web application framework, https://expressjs.com, accessed on Feb. 28. 2021
[29] mysql2 - npm, https://www.npmjs.com/package/mysql2, accessed on Feb. 28. 2021
[30] C. Staicu, M.T. Torp, M. Schäfer, A. Møller, and M. Pradel, "Extracting Taint Specifications for JavaScript Libraries," 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), pp. 198-209, Jun. 2020
[31] C. Staicu, M. Pradel, and B. Livshits, "SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS," Network and Distributed Systems Security (NDSS) Symposium 2018, Feb. 2018
[32] taser/new-lgtm-alerts.md, https://github.com/cs-au-dk/taser/blob/master/data/new-lgtm-alerts.md, accessed on Feb. 28. 2021
[33] mnutt/davros, https://github.com/mnutt/davros, accessed on Feb. 28. 2021
[34] FineUploader/server-examples, https://github.com/FineUploader/server-examples, accessed on Feb. 28. 2021
[35] giper45/DockerSecurityPlayground, https://github.com/giper45/DockerSecurityPlayground, accessed on Feb. 28. 2021
[36] spikebrehm/isomorphic-tutorial, https://github.com/spikebrehm/isomorphic-tutorial, accessed on Feb. 28. 2021
[37] AmpersandJS/ampersand, https://github.com/AmpersandJS/ampersand, accessed on Feb. 28. 2021
[38] halohalospecial/atom-elmjutsu, https://github.com/halohalospecial/atom-elmjutsu, accessed on Feb. 28. 2021
[39] nhn/tui-editor #1022, https://github.com/nhn/tui.editor/pull/1022, accessed on Feb. 28. 2021
[40] CVE-2019-1020008: stacktable.js before 1.0.4 allows XSS, https://www.cvedetails.com/cve/CVE-2019-1020008/, accessed on Feb. 28. 2021
[41] The exponential cost of fixing bugs, https://deepsource.io/blog/exponential-cost-of-fixing-bugs/, accessed on Mar. 25. 2021
[42] Flow: A Static Type Checker for JavaScript, https://flow.org, accessed on Mar. 25. 2021

〈저자소개〉

문 태 근 (Taegeun Moon) 학생회원
2020년 2월: 성균관대학교 소프트웨어학과 학사 졸업
2020년 3월~현재: 성균관대학교 소프트웨어학과 석사과정
〈관심분야〉 웹 어플리케이션 보안, 소프트웨어 분석

김 형 식 (Hyoungshick Kim) 종신회원
1999년 2월: 성균관대학교 정보공학부 학사
2001년 2월: KAIST 컴퓨터 과학과 석사
2012년 2월: University of Cambridge 컴퓨터공학과 박사
2013년 3월~현재: 성균관대학교 소프트웨어학과 부교수
〈관심분야〉 보안공학, 모바일 보안