자바스크립트에 특화된 프로그램 종속성 그래프를 이용한 표절 탐지

(Plagiarism Detection Using Dependency Graph Analysis Specialized for JavaScript)

김 신 형 † 한 태 숙 ††
(Shin-hyong Kim) (Taisook Han)

요 약 자바스크립트는 현재 웹 사이트, 웹 어플리케이션에서 가장 많이 사용되는 스크립트 언어 중 하나이다. 자바스크립트로 작성된 프로그램은 원본 프로그램 형태로 클라이언트에게 전송되므로 무단 복제, 도용에 쉽게 노출된다. 때문에 자바스크립트 프로그램의 도용을 탐지하기 위한 연구가 필요하다. 현재 일반적으로 프로그램 표절 탐지를 위해 사용되는 자동화 도구들의 경우 고수준의 표절 기법에 적절히 대응하지 못한다. 반면에 프로그램 종속성 그래프에 기반을 둔 기존 연구들의 경우 자바스크립트의 동적인 특징을 적절히 반영하지 못한다. 또한 지나친 일반화로 인해 일부 틀린 판정(false positive)을 보이며 대상 프로그램의 크기가 클 경우 탐지 속도에 문제를 보이고 있다. 본 논문에서는 자바스크립트에 특화된 프로그램 종속성 그래프(이하 JS PDG)와 이를 사용한 도용 탐지 기법을 제안하여 이러한 문제를 해결하고자 한다. 본 논문에서 제안하는 JS PDG는 세분화된 노드 타입을 가지고 있어 기존 PDG와 비교해 보다 정확한 그래프 간 비교를 할 수 있도록 하며 포함하고 있는 노드 타입에 따라 정의되는 JS PDG의 타입은 탐색 범위를 분할을 가능하게 해 전체 도용 탐지 속도가 개선 될 수 있도록 한다. 실험 결과 기존 PDG에서 나타나는 틀린 판정을 확인할 수 있었으며 PDG간 비교 횟수가 줄어들어 도용 탐지 속도가 개선됨을 확인할 수 있었다.

키워드 : 자바스크립트, 프로그램 종속성 그래프, 소프트웨어 도용 탐지, 복제 탐지

Abstract JavaScript is one of the most popular languages to develope web sites and web applications. Since applicationss written in JavaScript are sent to clients as the original source code, they are easily exposed to plagiarists. Therefore, a method to detect plagiarized JavaScript programs is necessary. The conventional program dependency graph(PDG) based approaches are not suitable to analyze JavaScript programs because they do not reflect dynamic features of JavaScript. They also generate false positives in some cases and show inefficiency with large scale search space. We devise a JavaScript specific PDG(JS PDG) that captures dynamic features of JavaScript and propose a JavaScript plagiarism detection method for precise and fast detection. We evaluate the proposed plagiarism detection method with experiment. Our experiments show that our approach can detect false-positives generated by conventional PDG and can prune the plagiarism search space.

Key words : JavaScript, Program Dependency Graph, Software Plagiarism Detection, Clone Detection

· 이 논문은 2010년도 정부(교육과학기술부)의 재원으로 한국과학재단의 지원을 받아 수행된 연구임(No. 2010-0000258)
· 본 연구는 지식경제부 및 정보통신산업진흥원의 대학 IT연구센터 지원사업의 연구결과로 수행되었음(NIPA-2010-C1090-1031-0004)

† 학생회원 : 한국과학기술원 전산학과
mywayin2sky@pllab.kaist.ac.kr
†† 종신회원 : 한국과학기술원 전산학과 교수
han@cs.kaist.ac.kr

논문접수 : 2010년 1월 12일
심사완료 : 2010년 3월 12일

정보과학회논문지: 소프트웨어 및 응용 제37권 제5호(2010.5)

1. 서 론

자바스크립트는 현재 웹 사이트나 웹 어플리케이션 개발을 위해 가장 많이 사용되고 있는 언어의 하나로 점차 복잡해지고 다양해져가는 웹 사이트와 웹 어플리케이션에서 많은 부분을 담당하고 있다. 최근 몇 년간 Ajax[1] 기술이 소개되어, 구글 지도[1], 문서도구[2]와 같이 이 기술을 사용한 웹 개발이 활발하게 이루어지고 있는 점 역시 자바스크립트의 사용을 더욱 활성화하는 역할을 하고 있다.

클라이언트 측에서 동작하도록 설계된 자바스크립트의 특성은 개발자에게 유연성을 제공하기도 하지만 한편 자신이 개발한 프로그램이라는 지적 재산을 보호하는 데는 어려움을 주고 있다. 현재 자바스크립트 프로그램이 도용되는 것을 막기 위해 압축, 난독화 등의 노력이 이루어지고 있지만 이렇게 변형된 프로그램은 수동, 자동으로 원본 프로그램에 가까운 형태로 복구가 가능하기 때문에 궁극적인 해결책이 되지 못한다.

따라서 자바스크립트 프로그램을 대상으로 도용 탐지를 위한 연구가 필요하다. 그러나 현재 도용 탐지를 위한 자동화 도구들[2-6]은 토큰에 기반을 둔 탐지 기법을 사용하므로 이들을 피할 수 있는 표절 기법들이 많이 있다. 예를 들어 도용한 프로그램의 실행문 순서를 바꾸거나 제어문을 대체하거나 불필요한 코드를 삽입했을 경우 이들 자동화 도구로는 탐지가 불가능하다.

한편, 프로그램 종속성 그래프(Program Dependency Graph, 이하 PDG라 칭함) 분석을 사용할 경우, 앞에서 언급한 고수준의 표절 기법이 사용된 프로그램에 대해서도 원본 프로그램과의 유사성을 찾아낼 수 있다. 그러나 기존 PDG 분석[7-9]을 사용하여 자바스크립트 프로그램간 도용 탐지를 할 경우, 몇 가지 문제점이 발견되었다.

```
S0: var str = prompt();
S1: var a = new Object();
S2: var a.x = new Object();
S3: var a.y = new Object();
S4: b = a[str];
```

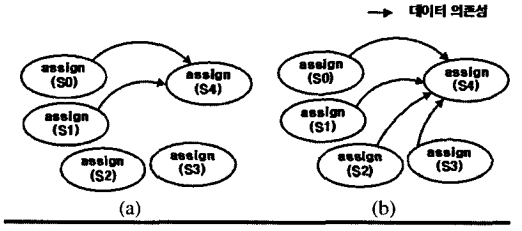

그림 1 자바스크립트의 동적인 특성의 예

```
(a)
S0:  function actOnUnload(){
S1:    var sParentLoc = document.location.toString();
S2:    var idx0 = sParentLoc.indexOf("preview");
S3:    var isPreview = (idx0 > -1) ? true : false;
S4:    if (isPreview)
S5:      window.opener.editor.setUnloadWarn(true);
S6:  }

(b)
S0:  function chkUserAgent(){
S1:    var nName = window.navigator.appName();
S2:    var idx = nName.indexOf("Explorer");
S3:    var IE = (idx0 > -1) ? true : false;
S4:    if (IE)
S5:      window.navigator.setUserAgent(IE);
S6:  }
```

(c)

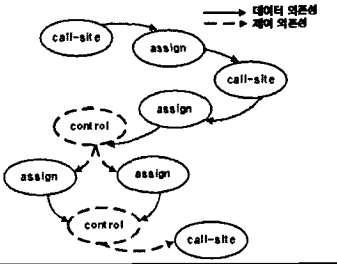

그림 2 기존 PDG 분석의 틀린 판정의 예

먼저 기존 PDG는 자바스크립트의 동적인 특징을 반영하지 못한다. 그림 1에 나타난 코드는 자바스크립트 포인터 분석(points-to analysis)을 위해 사용된 예이다[10]. 이를 기존 PDG로 표현하면 (a)와 같다. 그런데 S4의 a[str]은 동적인 시간에 a.x나 a.y를 가리킬 수 있다. 이는 자바스크립트의 동적인 특징으로 이러한 데이터 의존성(Data Dependency)은 기존 PDG에 반영되지 않는다.

다음으로 기존 PDG 분석을 사용한 도용 탐지는 일부 틀린 판정을 도출한다. 그림 2(a)와 (b)의 두 자바스크립트 프로그램을 대상으로 기존 PDG 분석 방법을 적용할 경우 (c)의 PDG가 동일하게 생성되어 도용으로 판단된다. 그러나 두 함수를 살펴보면 두 함수가 서로 다른 의미를 가지고 있는 것을 알 수 있다.

마지막으로 일반적으로 PDG 분석을 이용한 도용 탐지 방법은 탐색 범위(search space)가 방대할 경우 속도에 문제를 보인다고 알려져 있다[9]. 표 1은 실제 사용되는 자바스크립트 프로그램의 크기를 lcounter[3]를 이용해 측정한 도표이다.

표 1 자바스크립트 프로그램의 예

웹 페이지	줄 수	함수	설명
maps.google.com	5796	49	지도표시
www.screentoaster.com	8991	94	화면저장
instacalc.com	11587	21	계산기

1) 구글 지도, http://maps.google.com
2) 구글 문서도구, http://docs.google.com
3) lcounter, http://noeld.com/lcounter

표 1에 나타난 바와 같이 단일 페이지에 포함된 자바스크립트 프로그램의 크기가 결코 작지 않으므로 실제 웹 사이트가 포함하고 있는 자바스크립트 프로그램의 크기는 단일 페이지에 포함된 프로그램의 몇 배 혹은 몇 십 배의 크기를 갖게 되어 기존 PDG 분석을 이용해 웹 사이트 단위의 도용 탐지를 할 경우 속도에 문제를 보일 것으로 보인다.

본 논문에서는 앞에서 발견된 문제점들을 해결하기 위해 자바스크립트에 특화된 프로그램 종속성 그래프(이하 JS PDG라 칭함)와 이를 이용한 도용 탐지 기법을 제안한다. JS PDG는 자바스크립트의 동적인 특징을 반영하여 에지(edge)을 구성하며 노드(node)의 타입 정보를 가지고 있어 보다 정확한 자바스크립트 프로그램의 의미를 나타낼 수 있다. 또한 각 JS PDG가 포함하고 있는 특수 노드를 이용하여 한정된 탐색 범위를 제공함으로써 전체적인 도용 탐지의 속도를 향상 시킬 수 있는 방법을 제시한다.

2. 관련 연구

기존에 사용되고 있는 소프트웨어 표절 탐지 기법들을 대략 아래와 같이 분류할 수 있다.

1. 문자열 분석에 기반을 둔 탐지 기법
2. 토큰 분석에 기반을 둔 탐지 기법
3. 추상화된 문법 트리 분석에 기반을 둔 탐지 기법
4. PDG 분석에 기반을 둔 탐지 기법

먼저, 문자열 분석에 기반을 둔 탐지 기법은 식별자(Identifier)를 바꾸는 등의 저수준의 표절 기법이 적용된 프로그램에 대해서도 표절을 탐지해내지 못한다[9].

다음으로 도용 탐지를 위한 자동화 도구들[2-5]은 대부분 토큰에 기반을 둔 방법이다. 때문에 프로그램 실행문의 순서를 변경하거나 제어문의 종류를 대체하거나 불필요한 코드를 삽입하는 등의 표절 기법에는 우수한 탐지 능력을 보여주지 못한다. 또한 추상화된 문법 트리 분석[11,12]을 사용할 경우에도 동일한 문제점을 보인다.

웹 어플리케이션의 함수들의 복제 탐지를 위해 사용된 기법[13] 역시 토큰에 기반을 둔 연구의 일종으로 앞에서 언급한 고수준의 표절 기법에는 적절하게 대응하지 못한다.

PDG를 통한 프로그램의 분석은 컴파일러의 최적화를 위해 처음 제안되었으며[14] 이를 이용한 도용 탐지에 관한 연구들도 다수 이루어졌다[7-9]. 그러나 이러한 연구에 사용된 기존 PDG는 자바스크립트 프로그램들을 대상으로 그대로 사용하기 어렵다. 기존 PDG의 경우 자바스크립트의 동적인 특성을 반영하지 않으며 표면적인 구조가 동일한 두 프로그램에 대해 도용이라는 틀린 판정을 나타내며 이를 이용한 도용탐지는 탐색 범위가 넓을 경우 일반적으로 속도에 문제를 나타내기 때문이다.

3. JS PDG의 구성상 특징

PDG는 함수 단위의 프로그램을 그래프 형태로 나타낸 것으로 기존 PDG에서 노드는 단순한 형태의 실행문(statement) 또는, 표현문(expression)등으로 이루어진다. 에지는 노드들이 가지고 있는 제어 의존성(Control Dependency)과 데이터 의존성(Data Dependency)에 따라 생성된다[14]. 본 논문에서 제안하고 있는 JS PDG의 구성상 특징은 다음과 같다.

3.1 JS PDG에서의 노드와 에지의 구성

JS PDG에서의 각 노드는 기존 PDG에서 사용되는 노드를 바탕으로 보다 정확한 프로그램의 의미를 나타낼 수 있도록 특수 노드를 추가했다. 전체 노드 타입(node type)은 표 2에 표시했다. 특수 노드_ɣ라는 것은 표 3에서 정의한 특정 객체나 값을 포함하고 있는 노드에 한해 그 객체나 값이 의미하는 ɣ라는 종류로 분류한 것이다.

표 2 JS PDG의 전체 노드 타입

노드 타입(node type)		설명
일반	함수 정의문	함수 이름
	함수 호출문	함수 호출
	제어문	분기, 반복문의 조건
	선언문	변수, 객체 선언
	할당문	할당문(assign)
	표현문	기타 일반 표현문
특수	함수 정의문_ɣ	노드 타입_ɣ를 포함하고 있는 문
	함수 호출문_ɣ	
	제어문_ɣ	
	선언문_ɣ	
	할당문_ɣ	
	표현문_ɣ	

표 3 JS PDG의 특수 노드

특수노드(_ɣ)	설명
노드타입_w	window 객체가 사용된 노드
노드타입_d	document 객체가 사용된 노드
노드타입_p	prototype 객체가 사용된 노드
노드타입_r	정규표현을 값으로 갖는 변수가 사용된 노드
노드타입_f	함수를 값으로 갖는 변수가 사용된 노드

JS PDG에서 에지를 구성할 때에는 동적인 참조에 대한 고려가 필요하다. 여기에서 동적 참조란 다른 범용 언어들과 구별되게 자바스크립트가 가지는 동적인 특징으로 인해 생성되는 에지를 말한다. 본 논문에서는 동적 참조를 생성하기 위해 고려한 자바스크립트의 특징들은 [] 연산자와 프로토타입 객체이다.

JS PDG에서 일반적으로 데이터 의존성 에지를 생성

하기 위해서는 데프-유즈 체인(def-use chain)[15]을 사용하는데 입력 프로그램의 추상화된 문법 트리로부터 임의의 변수가 정의되거나 사용된 노드의 목록, 즉 데프 리스트(def list)와 유즈 리스트(use list)를 구성한 후 유즈 리스트의 각 원소에 대해 라이브(live)한 데프 리스트의 원소를 찾아 에지를 구성하게 된다. 데프 리스트의 원소가 라이브 하다는 것은 데프 리스트의 원소와 유즈 리스트의 원소를 포함하고 있는 노드들 사이에 동일 원소에 대한 다른 정의가 존재하지 않으며 실행 가능한 패스(path)가 존재하는 것을 말한다. 위에서 언급한 두 가지 특징들에 대해 동적인 참조를 생성하기 위한 구체적인 방법은 그림 3과 같다.

예를 들어 그림 1의 코드 S4에서 [] 연산자를 통해 객체 a의 str이라는 속성 즉, a[str]이란 변수의 값을 갱신할 경우 실제로 a[str]이 가리키는 객체는 정적인 시간에 결정되지 않는다. a[str]이 가리키는 객체는 a.x가 될 수도 있고 a.y가 될 수도 있으며 새로운 객체가 될 수도 있다. 따라서 그림 1의 (b)와 같이 동적인 참조를 반영하는 에지를 추가해야 한다.

자바스크립트에서 프로토타입(prototype) 객체의 속성이 생성되거나 수정되는 경우에도 동적인 참조가 생성된다. 그림 4의 코드를 기존 PDG로 구성할 경우 (a)와 같다. 그러나 정확한 프로그램의 의미를 표현하기 위해서는 S4에서 Person의 prototype에 새롭게 추가된 name이란 속성이 S5에서 사용됨으로 생성되는 데이터 의존성을 (b)와 같이 나타내어야 한다.

1. [] 연산자에 의해 생성되는 동적 참조

D: def list를 나타내는 멀티셋(multiset)
U: use list를 나타내는 멀티셋(multiset)
Val(γ): 변수 γ가 가지는 값
N(δ): δ를 포함하고 있는 노드

```
for ∀α,β. α[β] ∈ U
  if β is variable
    if Val(β) is a string constant
      for each α.Val(β) ∈ D
        if α.Val(β) is live
          make_edge N(α.Val(β)) to N(α[β])
    else // means Val(β) isn't determined statically
      for each ∀ψ. α.ψ ∈ D
        if α.ψ is live
          make_edge N(α.ψ) to N(α[β])
```

2. 프로토타입 객체에 의해 생성되는 동적 참조

```
for ∀α,β. α.β ∈ U
  for ∀ω. ω is construct function of α
    for each ω.prototype.β ∈ D
      if ω.prototype.β is live
        make_edge N(ω.prototype.β) to N(α.β)
```

그림 3 JS PDG에서의 동적 참조 생성

```
S0: function Person(){
S1:     this.gender = "male";
S2: }
S3: tom = new Person;
S4: Person.prototype.name = "Thomas";
S5: tommy = tom.name;
```

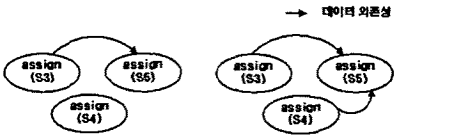

그림 4 JS PDG에서의 동적 참조 생성의 예

3.2 JS PDG에서의 특수 노드

기존 PDG 분석 기법에서는 각 노드를 구성하는 변수들의 데이터 타입(Data type)을 제거한 후 노드를 구성한다. 그 이유는 C, C++, Java와 같은 범용 언어의 경우 상호 대체하여 사용 가능한 데이터 타입들이 많이 존재하기 때문에 이러한 데이터 타입의 대체가 표절 기법으로 사용될 수 있기 때문이다.

자바스크립트 언어의 경우 다른 타입으로 대체하여 사용하기 어려운 독특한 객체 타입, 데이터 타입들이 존재한다. 본 논문에서는 이러한 객체나 데이터를 포함한 노드를 특수 노드로 정의했다.

이렇게 정의된 특수 노드는 표절 탐지 시 제어 의존성, 데이터 의존성 외에 추가적인 정보를 제공하는 역할을 한다. 전체 특수 노드는 표 3에 표시하였으며 각 특수 노드에 대한 자세한 설명은 다음과 같다.

3.2.1 호스트 객체(Host Object)

자바스크립트는 본래 독자적으로 사용하기 보다는 다른 어플리케이션에 포함하여 사용하도록 개발되었다. 예를 들어 웹 브라우저에서는 윈도우(Window)와 도큐먼트(Document)를 제어하기 위해 자바스크립트 고유의 객체가 아닌 브라우저 객체 모델(Browser Object Model), 도큐먼트 객체 모델(Document Object Model)을 구현한 객체들을 함께 사용하고 있다. ECMAScript 명세서[16]에서는 이러한 객체들을 호스트 객체(Host Object)로 정의하고 있다. 이러한 호스트 객체 중 윈도우 객체와 도큐먼트 객체의 경우 웹 사이트나 웹 어플리케이션을 구현하는 자바스크립트 프로그램에서 필수적으로 사용해야하는 객체이다. 사용자의 입력을 받거나, 수행 결과를 윈도우나 도큐먼트에 출력하기 위해서는 반드시 이 객체들을 사용해야하기 때문이다. 이 두 가지 객체들은 다른 객체들로 대체되기 어렵기 때문에 특수 노드로 분류했다.

3.2.2 프로토타입 객체(Prototype Object)

자바스크립트는 객체지향언어이며 프로토타입을 기반

으로 한 상속 방법을 사용한다. 간단히 설명하면 자바스크립트의 모든 객체가 기본적으로 가지고 있는 prototype이라는 속성에 생성자 함수를 할당함으로써 상속관계가 형성된다. 또한 동적인 시간에 각 객체의 프로토타입에 속성을 추가하거나 변경하는 일이 가능하다. 프로토타입 객체는 다른 객체로 대체할 수 없기 때문에 특수 노드로 분류했다.

3.2.3 기타 특수 노드

자바스크립트의 변수는 명시적인 데이터 타입을 가지고 있지 않다. 동적인 시간에 할당된 값에 따라 데이터 타입을 가지게 된다. 이러한 자바스크립트 변수가 가질 수 있는 값의 종류는 매우 다양한데 그 중 정규 표현(Regular Expression)과 함수 표현(Function Expression)의 경우 다른 데이터 타입으로 대체하여 사용하기가 매우 어렵다. 따라서 위의 두 가지 타입의 변수를 특수 노드로 분류했다.

4. 자바스크립트 표절 탐지 기법

본 논문에서 제안하는 자바스크립트 표절 탐지 기법의 동작은 다음과 같다. 먼저 원본 프로그램 P와 도용이 의심되는 프로그램 P'를 구성하는 함수들을 대상으로 함수 당 하나의 JS PDG를 생성하여 두 개의 JS PDG 집합을 구성한다. 이때 표 3에 명시된 특수 노드를 포함한 함수들의 JS PDG는 표 4에 나타난 바와 같이 분류된다. 복수개의 특수 노드를 포함한 함수의 경우 복수개의 타입을 가지게 된다.

표 4 JS PDG에서의 함수 분류

포함된 특수 노드	특수 JS PDG 분류(_y)
노드타입_w	function_W
노드타입_d	function_D
노드타입_p	function_P
노드타입_r	function_R
노드타입_f	function_F

JS PDG 생성이 끝나면 원본 프로그램에서 생성된 JS PDG 집합 G의 모든 원소에 대해 도용이 의심되는 프로그램 P'로부터 생성된 G'의 모든 원소와의 서브그래프 동형 확인[17,18]을 통해 표절 여부를 판단한다. 전체 알고리즘은 그림 5에 표시했다. 본 논문에서 제안하는 자바스크립트 표절 탐지 기법의 특징은 다음과 같다.

4.1 탐색 범위 분할을 통한 속도 개선

일반적으로 서브 그래프 동형 확인을 통한 표절 탐지에서는 원본 프로그램과 도용이 의심되는 프로그램 두 개의 프로그램에서 생성된 모든 PDG 쌍에 대해 즉, |G| × |G'| 쌍에 대한 비교가 필요하다. 그러나 JS PDG를 사용할 경우 탐색 범위의 분할이 가능하여 전체 탐색 범위가 줄어들 수 있다.

```
INPUT  : G, G'
G  = 프로그램 P 로부터 생성된 JS PDG들의 집합
G' = 프로그램 P'로부터 생성된 JS PDG들의 집합
OUTPUT : (g, g') // 표절 가능성이 있는 JS PDG 쌍

for g ∈ G
  for g' ∈ G'
    if g is function_Y
      for (g' ∈ G' and g' is function_Y)
        if (g, g') is subgraph-isomorphic
          return (g, g')
    else if g is normal function
      for (g' ∈ G')
        if (g, g') is subgraph-isomorphic
          return (g, g')
```

그림 5 자바스크립트 표절 탐지 알고리즘

기존 PDG를 사용할 경우, 원본 프로그램으로부터 생성된 모든 함수들에 대해 도용이 의심되는 프로그램으로부터 생성된 모든 PDG와의 서브그래프 동형 확인을 해야 한다. 즉, 각 프로그램으로부터 생성되는 PDG 수의 곱이 전체 탐색 범위가 된다.

JS PDG를 사용할 경우, 원본 프로그램으로부터 생성된 함수가 일반 함수일 경우 기존 PDG를 사용했을 경우와 마찬가지로 도용이 의심되는 프로그램으로부터 생성된 모든 PDG와의 비교가 필요하다. 그러나 원본 프로그램으로부터 생성된 함수가 특수 함수일 경우 도용이 의심되는 프로그램으로부터 생성된 PDG 중 동일한 종류의 특수 PDG에 한해서만 서브 그래프 동형 확인을 하기 때문에 탐색 범위가 줄어들게 된다.

이는 일반 함수의 경우 도용 시 특수 노드가 삽입되어 함수 타입이 변경될 수 있기 때문에 전체 함수들에 대해서 서브 그래프 동형 확인을 해야 하지만 특수 함수의 경우 3.2절에서 설명한 바와 같이 특수 노드를 제거하여 도용하는 것이 불가능하기 때문에 원본 프로그램에서 가졌던 특수 함수 타입을 그대로 가지고 있기 때문이다.

4.2 정확도 개선

본 논문에서 제안하는 자바스크립트 표절 탐지 기법은 기존 PDG를 기반으로 하는 시스템에서 노드 타입의 지나친 일반화로 인해 틀린 판정으로 도출되는 대상에 대해서도 분별할 수 있는 능력을 가지고 있다.

서로 같은 제어 의존성과 데이터 의존성을 가지고 있더라도 노드의 데이터 타입이 각각 특수 점점과 일반 노드 또는 특수노드_w와 특수노드_d와 같이 서로 다를 경우 그것을 분별해 낼 수 있다. 예를 들어 그림 2에서 두 함수의 S1에 의해 생성된 두 노드의 타입은 각각 assign_w와 assign_d가 되므로 결과적으로 서로 다른 프로그램으로 판단될 수 있다.

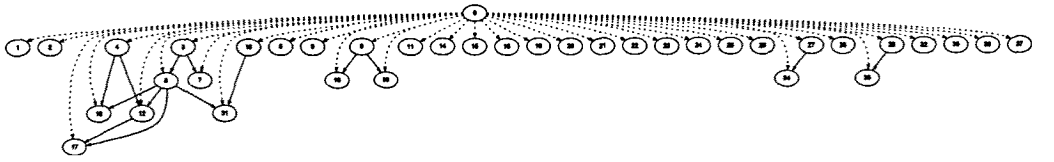

그림 6 생성된 JS PDG의 예

5. 실험 및 평가

관련연구에서 언급한 바와 같이 일반적으로 PDG는 현재까지 도용 탐지에 사용된 기타 연구들에 비해 비교적 우수한 성능을 나타낸다고 알려져 있다. 따라서 우리는 JS PDG의 성능을 평가하기 위한 실험의 중점을 기존 PDG 분석과의 비교에 두었다.

즉, JS PDG는 기본적으로 기존 PDG와 동일한 형태의 노드와 에지를 구성하게 된다. 노드 타입이라는 추가 정보를 가지고 있다는 점을 제외하면 노드와 데이터 의존성 에지가 동일한 형태로 구성되기 때문에 기존 PDG가 프로그램의 실행문 순서를 바꾸거나 제어문을 대체하거나 불필요한 코드를 삽입하는 등의 표절 기법을 사용해도 본래 그래프 형태가 유지됨으로써 저항력을 갖는 것과 마찬가지로 JS PDG 역시 본래의 그래프 형태를 유지한다. 따라서 결국 PDG 분석이 다른 분석 방법들에 비교해 우수하다고 평가받는 변별력이나 저항성에 대한 검증은 불필요하다고 생각된다.

본 실험에서는 4.1과 4.2에서 언급한 JS PDG가 가지고 있는 특징 즉, 속도와 정확도 개선에 대한 검증을 하고자 하였다.

전체적인 과정을 설명하면, 임의의 자바스크립트로부터 기존 PDG와 JS PDG를 추출할 수 있는 시스템을 각각 독립적으로 구현하고 기존 PDG나 JS PDG의 집합으로 부터 생성된 PDG들을 입력받아 서브 그래프 동형 확인을 할 수 있는 프로그램을 작성했다. 다음으로 원본 프로그램과 도용이 의심되는 프로그램으로 사용할 도용 프로그램을 선정하여 기존 PDG와 JS PDG를 사용하여 도용 탐지를 시행한 후 그 결과에 대해 살펴보았다. 구체적인 내용은 다음과 같다.

5.1 기존 PDG와 JS PDG의 생성

먼저, 임의의 자바스크립트 프로그램으로부터 기존 PDG와 JS PDG를 추출해 낼 수 있는 시스템을 각각 구현했다. 기존 PDG 분석을 자바스크립트에 적용한 사례가 없어 직접 구현했는데 spidermonkey[4]를 사용하여 자바스크립트 프로그램의 추상화된 문법 트리를 생성한 후 그것으로부터 PDG를 생성하는 프로그램을 작성했다. 기존 PDG 분석 구현 시의 노드 타입은 표 2의 JS PDG에서 특수 노드를 제외한 일반 노드로 구성하였으며 JS PDG의 경우 3장 JS PDG의 구성상 특징에 설명한 바와 같이 기존 JS PDG와의 구별되는 특징을 반영하여 각각 구현하였다.

그림 6은 생성된 JS PDG의 예로 prototype 1.5.1의 Ajax.Request.prototype 함수 내부에 정의된 request라는 함수의 JS PDG이다. 데이터 의존성 에지가 실선으로 나타나 있다.

5.2 서브 그래프 동형 확인

임의의 프로그램으로부터 기존 PDG와 JS PDG의 집합을 생성한 후 서브 그래프 동형을 확인하기 위해 다음과 같은 작업을 하였다. 여러 가지 알고리즘(algorithm)을 선택하여 그래프 동형 확인을 할 수 있도록 지원하고 있는 C++ 라이브러리인 VFLib[5]을 사용하되 프로그램의 일부를 수정하여 원본 함수를 구성하는 PDG 전체 노드의 80%이상이 포함될 경우 서브 그래프 동형으로 판단하도록 하고 원본 프로그램의 각 함수에 대해 도용 프로그램의 모든 함수들에 대해 서브 그래프 동형 확인을 하도록 했다.

5.3 대상 프로그램의 선정

다음으로 대상 프로그램의 선정은 다음과 같다. 현재 많이 사용되는 자바스크립트 라이브러리(library) 중 3개를 선택한 후 각 프로그램에 대해 2개의 버전(version)을 대상 프로그램으로 선정하여 하위 버전을 원본 프로그램으로, 상위 버전을 도용이 의심되는 도용 프로그램으로 가정했다. 일반적으로 상위 버전에서 하위 버전에서 사용하던 함수를 그대로 사용하거나 약간의 변경을 하되 본래의 기능 그대로 사용하는 경우가 많이 있기 때문에 그러한 함수 쌍을 원본과 도용이 의심되는 함수 쌍으로 찾고자 하는 것이다.

다음으로 PDG의 분별력을 위해서 각 프로그램의 함수들로부터 생성된 PDG 중 노드의 수가 10개 이상이고 데이터 의존성 에지가 5개 이상인 함수들만을 실험 대상으로 선정 했다. 그 이유는 노드와 에지가 일정 수 이상 되면 서브 그래프 동형 확인 시 틀린 판정으로 동형이라고 판단하게 되는 경우가 적어지기 때문이다. [9]와 같은 경우 PDG간의 동형 확인 시 노드 5개 이상인 PDG를 대상으로 선정했는데 본 논문에서는 보다 엄격

4) http://www.mozilla.org/js/spidermonkey/

5) http://amalfi.dis.unina.it/graph/db/vflib-2.0/doc/vflib.html

표 5 실험 대상 프로그램

각 프로그램의 함수 개수 측정	하위 버전		상위 버전	
	전체	대상	전체	대상
jquery[6](1.2.6, 1.3.2)	274	40	271	54
prototype[7](1.5.1, 1.6.0)	346	55	408	72
mochikit[8](1.3.1, 1.4.2)	440	92	744	151

표 6 실험 대상 프로그램의 함수 분류

	하위 버전			상위 버전		
	전체	특수 함수	%	전체	특수 함수	%
jquery	40	13	32.5%	54	15	27%
prototype	55	7	12%	72	9	12%
mochikit	92	13	14%	151	27	17%

한 조건하에서도 기존 PDG의 틀린 판정이 나타나는가를 살펴보았다.

예를 들어 jquery의 경우 1.2.6 버전은 전체 함수 274개 중 40개의 함수가 대상 함수가 되며 1.3.2 버전은 54개의 함수가 도용 탐지를 위한 대상 함수가 된다. 각 프로그램의 대상 함수 개수는 표 5와 같다.

JS PDG를 사용하여 전체 대상 프로그램을 분석한 결과는 표 6과 같다. 표 5에서 선택한 대상 함수 가운데 표 4에서 분류한 특수 함수에 해당하는 함수들의 수와 백분율이 나타나 있다.

5.4 탐색 범위 분할을 통한 속도 개선

기존 PDG와 JS PDG를 각각 사용하여 도용탐지를 시행해 보았는데 탐지 과정에서 나타난 각 시스템의 서브 그래프 동형 확인을 위한 비교 횟수는 표 7과 같다. 전체 함수 대비 특수 함수의 비율이 하위, 상위 버전 각각 32.5%, 27%로 나타난 jquery의 비교 횟수는 JS PDG를 사용했을 경우 기존 PDG를 사용했을 경우와 비교해 약 29%가 줄어든 것을 확인할 수 있다. 표 7의 비교 횟수 개선 비율에서 나타난 바와 같이 전체 함수 대비 특수 함수의 비율이 높을수록 비교 횟수의 개선 비율 역시 높게 나타나는 것을 알 수 있다.

JS PDG를 사용했을 경우 기존 PDG에 비해 전체 비교 횟수가 줄어든 이유는 원본 함수들 중 특수 함수로 분류된 함수의 경우, 도용 함수들 중 동일한 종류의 특수 함수로 분류된 함수들만을 비교 대상으로 삼아 탐색 범위가 줄어들었기 때문이다. 서브 그래프 동형확인을 위한 PDG간 비교 횟수가 줄어들었다는 것은 잠재적으로 도용 탐지에 소요되는 속도 또한 개선될 수 있다는 것을 의미한다.

6) http://jquery.com/
7) http://www.prototypejs.org/
8) http://mochikit.com/

표 7 탐색 범위 분할을 통한 속도 개선

	특수 함수 비율		구분	전체 비교 횟수	비교 횟수 개선 비율
	하위	상위			
jquery	32.5	27	PDG	2160	29%
			JS PDG	1530	
prototype	12	12	PDG	3960	12%
			JS PDG	3472	
mochikit	14	17	PDG	13892	13%
			JS PDG	12064	

5.5 정확도 개선

기존 PDG 분석과 JS PDG 분석을 사용하여 실험 대상 프로그램들에 대해 도용 탐지를 실시한 결과는 표 8과 같다. jqeury의 경우 기존 PDG는 전체 40개의 함수에 대해 21개의 함수를 도용이 의심된다고 판단했으며 JS PDG는 19개의 함수를 도용이 의심된다고 판단했다. 직접 확인한 결과 JS PDG가 도용이 의심된다고 판단한 19개의 함수는 기존 PDG에 의해 모두 도용이라고 판단되었으나 기존 PDG가 도용이라고 판단한 추가적인 2개의 함수는 기존 PDG과 JS PDG가 서로 다른 판단을 하였다. 마찬가지로 prototype과 mochikit의 경우에도 각 1개의 함수에 대해 서로 다른 판단을 하였다.

기존 PDG 분석에서는 도용이라고 판단했으나 JS PDG 분석에서는 도용이 아니라고 판단한 함수 쌍들은 표 9와 같다.

직접 확인해 본 결과 대부분의 경우 도용으로 판단된 두 함수는 서로 다른 이름과 기능을 가지고 있었으나 노드와 에지의 수가 적은 JS PDG들이 비교적 복잡한 구조의 PDG들 즉, 많은 노드와 에지를 가지는 JS PDG들에 우연히 서브 그래프 동형으로 판정받은 것을 확인할 수 있었다. 예를 들어 mochikit의 경우 노드가 17개이고 에지가 5개인 작은 PDG가 노드가 64개이고 에지가 26개인 큰 PDG에 서브 그래프 동형으로 판단된 것이다. 즉, 기존 PDG를 사용했을 경우 나타날 수 있는 틀린 판정의 실질적인 예라고 할 수 있는데 JS PDG의 경우 이들에 대해 도용이 아니라는 올바른 판정을 내린 것을 확인할 수 있었다.

표 8 노드 타입 정보를 이용한 정확도 개선

	전체함수	구분	도용이 의심된다고 판단한 함수
jquery	40	PDG	21
		JS PDG	19
prototype	55	PDG	33
		JS PDG	32
mochikit	92	PDG	70
		JS PDG	69

표 9 다른 판정이 나온 함수 쌍

	개수	함수 쌍
jquery	2	1.2.6 버전의 globalEval(), 1.3.2 버전의 ajax()
		1.2.6 버전의 dir(), 1.3.2 버전의 Sizzle.filter()
prototype	1	1.5.1 버전의 makePositioned(), 1.6.0 버전의 makePositioned()
mochikit	1	1.3.1 버전의 MochiKit.Base.update()의 두 번째 인수로 정의된 함수, 1.4.2 버전의 emitHTML()

표 10 JS PDG의 오류의 예

버전	코드
prototype 1.5.1	*if (**window**.opera)*
prototype 1.6.0	*if (**Prototype.Browser**.Opera)*

한 가지 예외로 prototype에서 틀린 판정으로 나타난 makePositioned 함수의 경우 브라우저 확인이라는 동일한 기능을 수행하기 위해 표 10과 같이 두 버전에서 각각 window객체와 Prototype이라는 서로 다른 객체를 사용했기 때문에 서로 다른 타입의 노드를 구성하게 되어 JS PDG에 의해 도용이 아닌 것으로 판단되었다. 그러나 직접 확인한 결과 두 함수가 거의 동일한 기능과 구조를 가지므로 도용으로 판단되어야 하며 결국 JS PDG의 오류라고 할 수 있다.

3.2에서 설명한 바와 같이 일반적으로 본 논문에서 정의한 특수 노드는 다른 객체나 값으로 대체하기 매우 어려운 특징을 가지고 있다. 위의 경우는 서로 다른 도큐먼트 객체 모델과 브라우저 객체 모델의 속성이 동일한 의미를 갖게 되기 때문에 발생한 오류라고 할 수 있다. 즉, 대체가 가능한 특수 노드가 발생한 것인데 차후 이들에 대한 표준화가 진행되어 동일한 의미를 가지는 객체들의 속성에 대해 같은 노드 타입을 갖도록 구성할 수 있을 것으로 판단되며 현재로서는 표준화된 도큐먼트 객체 모델과 브라우저 객체 모델이 없기 때문에 해결하기 쉽지 않은 문제이다.

이와 같이 특수 노드 객체를 대체 할 수 있는 경우는 극히 이례적인 경우라 할 수 있으며 일반적으로 특수 노드는 앞에서 언급한 바와 같이 대체가 어려운 고유의 특성을 가지고 있다.

6. 결론 및 연구 과제

본 논문에서는 특수 노드의 타입 정보를 포함하고 있는 JS PDG와 이를 이용한 도용 탐지 기법을 제안했다.

JS PDG에서 사용된 특수 노드들은 다른 객체, 값으로 대체 불가능한 특징을 가진 것들로 이들의 타입 정보를 이용해 보다 정확한 도용 탐지가 이루어질 수 있도록 추가적인 정보를 제공하는 역할을 한다. 실험 및 평가에서 살펴본 바와 같이 JS PDG 분석을 사용하여 기존 PDG 분석을 사용했을 경우의 틀린 판정 사례를 실제 사용되고 있는 자바스크립트 라이브러리 프로그램들로부터 찾아내었다.

JS PDG를 이용한 도용 탐지 기법에서는 각 JS PDG가 포함하고 있는 특수 노드 정보를 이용하여 탐색 범위를 분할함으로써 속도 개선을 했다. 프로그램을 구성하는 전체 함수에 대비해 특수 함수로 분류되는 함수들의 비중이 높을수록 탐색 범위 분할은 효과적으로 이루어진다.

우리는 향후 본 논문에서 제안하는 JS PDG와 이를 활용한 자바스크립트 도용 탐지 기법을 바탕으로 웹 사이트, 웹 어플리케이션 간의 도용 여부를 합리적인 시간 안에 탐지할 수 있는 자동화 시스템을 구축하고자 한다.

참 고 문 헌

[1] Paulson, L. D. et al., "Building rich web applications with Ajax," *In Computer*, vol.38. pp.14-17, 2005.

[2] J. W. Michael, "YAP3: improved detection of similarities in computer program and other texts," *SIGCSE Bulletin*, vol.28, pp.130-134, 1996.

[3] L. Prechelt, et al., "Finding plagiarisms among a set of programs with JPlag," *J. of Universal Computer Science*, vol.8, pp.1016-1038, 2002.

[4] T. Kamiya, et al., "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code," *IEEE TSE*, vol.28, no.7, pp.654-670, 2002.

[5] S. Schleimer, et al., "Winnowing: local algorithms for document fingerprinting," *In Proc. of ACM SIGMOD Int. Conf. on Mgmt. of Data*, 2003.

[6] S. Bellon, et al., "Comparison and Evaulation of Clone Detection Tools," *IEEE Trans. on Software Engineering*, vol.33, no.9, September 2007.

[7] J. Krinke, "Identifying similar code with program dependence graphs," *In Proc. WCRE'01. IEEE*, 2001.

[8] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," *In Proc. of the 8th Int. Sym. on Static Analysis*, pp.40-56, 2001.

[9] Liu, C. et al., "GPLAG: detection of software plagiarism by program dependence graph analysis," *In Proc. of the 12th ACM SIGKDD Int. Conf. on Knowledge discovery and data mining*, pp.872-881, 2006.

[10] Dongseok Jang et al., "Points-to analysis for JavaScript," *In Proc. of the 2009 ACM sym. on*

Applied Computing, pp.1930-1937.
[11] I. Baxter et al., "Clone Detection Using Abstract Syntax Trees," *In ICSM*, pp.368-377, 1998.
[12] R. Koschke et al., "Clone Detection Using Abstract Syntax Suffix Trees," *In Proc. WCRE'06. IEEE*, 2006.
[13] Lanubile F. et al., "Finding Function Clones in Web Applications," *In Software Maintenance and Reengineering, 2003. Proc. 7th European Conf.*, pp.379-386.
[14] Jeanne Ferrante et al., "The Program Dependence Graph and its use in optimtization," *In ACM Trans. on Programming Languages and Systems*, vol.9, no.3, pp.319-349, July 1987.
[15] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers: principles, techniques, and tools, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1986.
[16] ECMAScript Language Specification Edition 3
[17] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. of the Association for Computing Machinery*, vol.23, pp.31-42, 1976.
[18] Luigi P. Cordella et al., "A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol.26, no.10, pp.1367-1372, Oct. 2004.

김 신 형
2008년 한동대학교 학사 졸업. 2009년~현재 KAIST 전산학과 석사과정. 관심분야는 프로그램 분석, 프로그래밍 언어

한 태 숙
1976년 서울대학교 전자공학과 학사 1978년 KAIST 전산학과 석사. 1990년 Univ. of North Carolina at Chapel Hill 박사. 1991년~현재 KAIST 전산학 전공 교수. 관심분야는 프로그래밍 언어, 함수형 언어, 임베디드 시스템