Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.30, NO.4, Aug. 2020 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2020.30.4.573

자바스크립트 엔진에 대한 시맨틱 보존적 변이기반 퍼징*

오 동 현,[1†] 최 재 승,[1] 차 상 길[2‡]
[1,2]한국과학기술원 소프트웨어보안연구실(대학원생, 교수)

Semantics-Preserving Mutation-Based Fuzzing on JavaScript Interpreters*

DongHyeon Oh,[1†] JaeSeung Choi,[1] SangKil Cha[2‡]
[1,2]SoftSec Lab., Korea Advanced Institute of Science and Technology
(Graduate student, Professor)

요 약

퍼징은 입력값을 무작위로 생성해 소프트웨어를 테스팅하는 방법으로, 처음 고안된 이래로 다양한 방식의 퍼징이 연구되고 있다. 그중 변이기법을 적용한 퍼징은 확률에 따른 비트 반전이나 특별 값 치환과 같이 비교적 간단한 접근법을 사용함에도, 많은 버그를 발견해온 만큼 효율적인 방법이라고 할 수 있다. 하지만 인터프리터는 문법, 시맨틱이 올바른 입력값을 요구하기 때문에 일반적인 변이기법을 적용하기에는 어려움이 있다. 이에 본 연구에서는 동적 데이터 흐름 분석을 통해 변이기법을 인터프리터 퍼징에 적용할 수 있는 방법에 대해 제시하고자 한다. 본 연구에서 제시하는 JMFuzzer는 문법, 시맨틱의 올바름을 고려해 자바스크립트 인터프리터에서 오류 없이 정상적으로 동작하는 다양한 유형의 테스트케이스를 생성할 수 있다. 최종적으로 본 연구에서는 최신 버전의 자바스크립트 인터프리터에서 알려지지 않은 취약점들을 찾았으며, 이를 각 회사에 제보했다.

ABSTRACT

Fuzzing is a method of testing software by randomly generating test cases. Since its introduction, a variety of fuzzing techniques have been studied. Among them, mutation-based fuzzing is an efficient method that finds real-world bugs even though it uses a simple approach such as probabilistic bit-flipping and character substitution. However, the interpreter fuzzing has difficulty in applying general mutation techniques because the interpreter requires grammar and semantic correctness input values. In this paper, we present a novel mutation-based fuzzing on JavaScript interpreters with a dynamic data flow analysis. To this end, we implement JMFuzzer that can generate various types of mutated test cases that operate normally without runtime errors in JavaScript interpreter considering syntax and semantics. As a result, we found numerous unknown vulnerabilities in the latest JavaScript interpreters. We reported all of them to the vendors.

Keywords: Data flow analysis, Fuzzing, JavaScript, Mutation, Security

I. 서 론

자바스크립트 엔진은 해커들에게 인기 있는 공격 벡터로 사용되고 있다. 특히 자바스크립트 엔진이 급격히 발전함에 따라 자연스레 많은 버그가 만들어지고 있다. 예를 들어 Just-In-Time(JIT) 최적화 기법은 그 복잡성과 불안정성으로 유명하다[1].

따라서 자바스크립트 엔진 퍼징에 관한 연구에도

Received(02. 28. 2020), Modified(1st: 05. 07. 2020, 2nd: 07. 10. 2020), Accepted(07. 10. 2020)
* 본 논문은 2019년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (No.2019-0-01697, 블록체인 플랫폼 보안취약점 자동분석 기술개발).
† 주저자, zanywhale@kaist.ac.kr
‡ 교신저자, sangkilc@kaist.ac.kr(Corresponding author)

```
function opt(arr, proto) {
    arr[0] = 1.1;
    let tmp = {__proto__: proto};
    arr[0] = 2.3023e-320;
}

function main() {
    let arr = [1.1, 2.2, 3.3];
    for (let i = 0; i < 10000; i++) {
        opt(arr, {});
    }
    opt(arr, arr);
    print(arr);
}

main();
```

```
function opt(arr) {
    arr[0] = 1.1;
    this[0] = {};
    arr[0] = 2.3023e-320;
}

function main() {
    let arr = [1.1];
    for (let i = 0; i < 10000; i++) {
        opt.call({}, arr);
    }
    opt.call(arr, arr);
    print(arr);
}

main();
```

Fig. 1. CVE-2018-0834, CVE-2018-0837

관심이 급증하고 있는데, 주로 두 변이 기반 퍼징 혹은 생성 기반 퍼징 전략을 쓰고 있다. 변이 기반 퍼징은 주어진 자바스크립트 코드를 랜덤하게 수정하는 방식이며, 생성 기반 퍼징은 자바스크립트 문법과 같은 특정한 생성 기준에 따라서 새로운 자바스크립트 테스트케이스를 생성하는 방식이다.

특히 생성 기반 퍼징보다 변이 기반 퍼징이 유리한 측면이 있는데, 과거의 사례에서 작은 차이만으로도 다른 유형의 버그가 발생하는 경우가 존재한다. 과거 자바스크립트 엔진에서 발생한 취약점들인 CVE-2018-0834, 0837, 0840[2, 3, 4]은 서로 유사한 형태를 띠고 있다. 실제로 Fig.1. 과 같이 몇 개의 스테이트먼트 차이 이외에 전체적인 구조는 거의 같은 양상을 띠는 것을 확인할 수 있다. 하지만 두 버그는 다른 CVE 번호를 부여받은 다른 유형의 취약점이다. 이를 통해 기존에 취약하던 자바스크립트 코드를 약간 변경하는 것만으로도 다른 취약점을 찾을 수 있다는 것을 알 수 있다. 이러한 유형의 취약점을 찾기 위해서는 하나의 자바스크립트 코드를 기반으로 조금씩 변화를 주어야 한다. 따라서 같은 코드를 이용해 조금씩 변형을 주는 방식인 변이기법이 비슷한 모양새의 코드를 생성하기에 유리한 방법이다. 과거에도 Table 1. 과 같이 변이기법이 적용된 퍼징 방식을 사용한 연구들이 있었으나, 이는 부정확한 정보를 기반으로 퍼징을 시도했기 때문에, 코드의 시맨틱의 올바름이 깨져 런타임 오류가 발생하는 것을 제대로 처리하지 못했다.

이처럼 자바스크립트 엔진은 문법적으로 올바른 입력을 기대하기 때문에 기존의 대부분의 퍼저들은 문법적으로 유효한 테스트케이스를 생성하는 데 중점을 두고 있다. 예를 들어 jsfunfuzz[5]는 하드 코딩된 입력 생성 규칙을 기반으로 코드를 생성하는 생성 기반 퍼저로, jsfunfuzz의 초점은 문법적으로 올바른 테스트케이스를 만드는 데 있다. LangFuzz[6]는 문장 기반의 변이를 수행하여 구문의 정확성을 유지하면서 테스트케이스를 생성하는 변이 기반 퍼저이다. 두 퍼저에서 생성된 테스트케이스는 문법적으로 유효하지만, 여전히 시드는 런타임 오류를 가지고 있다.

따라서 몇몇 연구자들은 최근 구문 오류뿐만 아니라 런타임 오류를 발생시키지 않는 유효한 시맨틱을

Table 1. Compare Each JavaScript Fuzzer

	Type	Method
jsfunfuzz	Generation	Syntax-aware
LangFuzz	Mutation	Syntax-aware
CodeAlchemist	Generation	Semantic-aware
Montage	Mutation	Semantic-aware

갖는 테스트케이스를 생성하는 것의 중요성을 인식하기 시작했다. CodeAlchemist[7]는 생성 기반 퍼저로, 주어진 자바스크립트 시드 파일을 분석하여 코드 브릭을 만들고, 각 코드 브릭은 앞뒤로 필요한 변수와 그 타입 정보를 이용해 적절히 조립하여 유효한 자바스크립트 코드를 생성한다. Montage[8]는 자바스크립트의 구조적인 특성을 배우기 위해 코드의 AST를 신경망 언어 모델을 통해 모델은 만들고 그 모델을 이용해 변이를 수행하는 변이 기반 퍼저이다. Montage는 테스트케이스를 생성하기 위해 자주 관찰되는 구조적 패턴을 유지하면서 시드를 변이한다.

두 퍼저는 자바스크립트 엔진 퍼징의 효율성을 크게 향상 시키고 있지만, 여전히 몇 가지 문제점들이 있다: (1) CodeAlchemist는 각 테스트케이스에 대해 자바스크립트 코드를 처음부터 구조화해야 하므로 기본적으로 무한한 입력 공간을 탐색해야 한다; (2) Montage는 언어 모델의 정밀도의 부족으로 인해 변이된 코드에서 런타임 오류가 자주 발생한다.

본 논문에서 우리는 테스트케이스의 시맨틱을 유지하면서도 런타임 에러를 최소화하는 변이 기반의 퍼저인 JMFuzzer를 제안한다. JMFuzzer는 무한한 입력 공간 문제에 대해 탄력성을 가지면서 의미있는 테스트케이스를 생성할 수 있다. 시드가 주어지면 먼저 use, def를 분석하고 이후 데이터 동적 흐름 분석을 통해 각 변수의 타입과 Available, Liveness에 대한 정보를 분석한다. 이후 분석된 정보를 기반으로 자바스크립트 코드의 시멘틱의 유효성을 유지하면서 자바스크립트 코드의 스테이트먼트를 한 개를 하나 이상의 다른 스테이트먼트로 바꾸는 변이 기반의 퍼징을 수행한다. 최종적으로 이와 같은 방법을 통해 기여할 수 있는 점은 다음과 같다.

- 본 연구를 통해 시드의 시맨틱을 보존하며 변이 방식을 자바스크립트 엔진 퍼징에 적용할 수 있는 방법을 제시하였으며 그 효용성을 보였다.
- 본 연구에서는 자바스크립트 엔진 버그를 찾기 위해 동적 데이터 흐름 분석을 적용해 자바스크립트 퍼징에 변이 방식을 적용한 JMFuzzer를 설계 및 구현했다.
- JMFuzzer를 통해 메이저 브라우저에서 사용되는 자바스크립트 엔진에 대한 퍼징을 수행했고, 그 결과 이전에 보고되지 않은 총 5개의 버그를 발견해 이를 각 회사에 보고했다.

본 논문의 2장에서는 관련 연구, 3장에서는 전반적인 시스템의 개요와 설계를 소개한다. 4장에서는 JMFuzzer의 실험 및 평가를 위해 기존의 state-of-the-art 퍼저인 CodeAlchemist, jsfunfuzz 및 Montage와 비교해 평가한 내용에 관해서 기술한다. 마지막으로 5장에서는 본 논문의 결론을 짓는다.

II. 관련 연구

퍼징은 굉장히 빠르게 발전하고 있는 연구 분야로, 대단히 많은 방식의 퍼징 연구가 이루어져 왔다[9]. 특히 본 연구에서 차용한 방법인 변이 기반의 퍼징에는 BFF[10], Radamsa[11], AFL[12, 13, 14]와 같은 관련 도구들이 존재하며, 놀라울 정도로 좋은 성과를 보인 방법들 또한 연구되어왔다[15, 16].

일반적인 변이 기반의 퍼징 이외에도 자바스크립트 퍼징 관련 연구들이 존재한다. Fuzzilli[17]는 비교적 최근에 공개된 state-of-the-art 자바스크립트 퍼저이다. 미리 정의된 fuzzIL이라는 중간 언어로 타입을 미리 정의해둔 뒤, 유효한 시맨틱 자바스크립트 코드를 생성해 런타임 에러를 없앤다. 이는 커버리지를 기반으로 퍼징을 수행하는 그레이박스 퍼징이다. 하지만 자바스크립트 코드를 fuzzIL로 변경할 수는 없으므로 유용한 시드 파일인 자바스크립트 시드 파일과 CVE 코드를 이용할 수 없다는 단점이 있다. 이외에도 Skyfire, TreeFuzz[18, 19]와 같이 자바스크립트 언어의 시맨틱을 확률적 모델을 통해 변이를 시도한 연구도 있었으며, LangFuzz[3]를 개선한 IFuzzer[20] 등 다양한 사전 연구가 있었다.

III. 개요 및 설계

본 연구의 궁극적인 목표는 자바스크립트 엔진 퍼징 시 변이기법을 적용했을 때 런타임 오류를 최소화해 동작하는 자바스크립트 코드를 생성하는 것이다. 일반적으로 자바스크립트 코드에서 각 타입 정보와 특정 시점에서 사용할 수 있는 변수, 그리고 변이되는 시점 이후로 사용되는 변수에 대한 정보를 모르고 있는 상태에서 변이한다면 런타임 오류가 발생할 확률이 높다. 따라서 변이를 수행했을 때에도 런타임 오류 없이 정상적으로 동작하는 코드를 만들기 위해

서는 이러한 정보들을 파악하고 있어야 한다. 이러한 정보들은 동적 데이터 흐름 분석을 통해 얻을 수 있으며, 이러한 정보를 통해 런타임 오류의 발생이 적도록 변이기법을 적용할 수 있다. 이 정보는 use, def 분석과 타입 정보를 동적으로 알아내는 방법을 이용해 최종적으로 available, liveness 정보를 아는 것을 목표로 한다.

3.1 CodeBase

본 연구에서 사용된 CodeBase는 동적 데이터 흐름 분석을 통해 얻은 정보인 available, liveness 정보를 스테이트먼트와 함께 가지고 있는 시드로 정의한다.

Available이란 각 스테이트먼트 이전에 미리 선언되어 있어서, 사용할 수 있는 변수나 함수를 의미한다. 본 연구에서는 해당 변수 및 함수의 이름과 타입 정보를 모두 available 정보에 저장한다. 이를 계산하기 위해서는 특정 변수나 함수가 선언될 때마다 available 셋에 더해가는 방식을 사용하며, delete 등으로 undefined 상태가 되면 available 셋에서 빠지게 된다. 추후에 스테이트먼트를 변이시킬 때는, available 집합에 들어있는 변수 및 함수만을 사용하도록 변이시켜야 한다.

Liveness는 변경될 스테이트먼트 이후에 실행되는 코드에서 사용될 변수인지를 알려준다. 이 정보가 필요한 이유는, 만약 변이될 코드가 뒤에서 사용되어야 하는 선언에 해당한다면, 변이될 스테이트먼트에서는 이후로 사용될 변수에 대한 선언이 필요하기 때문이다. 따라서 liveness 정보를 통해 이후에 해당 변수가 사용되는지에 대한 정보를 확인할 수 있으며 이를 통해 참조에러를 사전에 방지할 수 있다.

Fig.2. 는 실제 데이터 흐름 분석을 수행해 available, liveness 정보가 시드와 함께 저장된 CodeBase를 도식화해놓은 그림이다. 직관적인 이해를 위해 타입 정보는 적혀있지 않다. 스테이트먼트를 하나씩 실행시킴에 따라 available 셋에는 foo

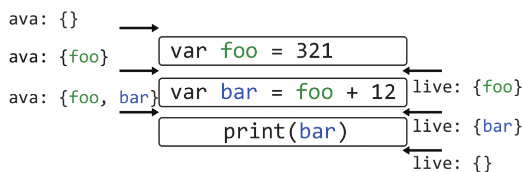

Fig. 2. Example of dynamic data flow analysis

와 bar라는 변수가 쌓이게 되며, 실행 흐름을 거꾸로 분석하며 liveness를 계산하며 liveness 셋에 더한다. 위 예제에서는 두 번째 스테이트먼트에서 foo 변수를 사용하고 있으므로 liveness 셋에 foo 변수가 포함되며, 이후에는 더는 사용되지 않는 변수이기 때문에 liveness 셋에 빠진 것을 볼 수 있다. 또, 세 번째 스테이트먼트에서 bar 변수를 사용하고 있으므로 liveness 셋에 bar가 추가된 것을 볼 수 있으며 마지막 스테이트먼트는 항상 liveness 셋이 공집합인 것을 확인할 수 있다.

3.2 구조

Fig.3. 은 JMFuzzer의 전체 시스템의 동작 과정 및 구조를 전반적으로 설명해 놓은 그림이다. 먼저 Seed Parser에서는 시드를 Esprima[21] 툴을 이용해 AST로 파싱한다. 이후 Constraint Analyzer에서는 타입을 분석할 수 있도록 코드 중간중간에 각 스테이트먼트 별 타입 분석용 코드를 삽입하고 각 시드를 실행하여 타입 분석을 수행한다. 분석 결과 타입 정보가 포함된 시드를 CodeBase라 하며, CodeBase는 사용하는 변수, 함수의 이름과 각각의 타입 정보 그리고 이를 가공한 정보인 available, liveness 셋을 포함한다. 이때, CodeAlchemist에서 사용하는 CodeBrick을 만드는 작업도 같이 진행하는데, CodeBrick은 스테이트먼트에 타입 정보가 있는 코드의 조각들로, CodeBrick의 타입을 알아내는 과정은 CodeBase와 다른 개념이기 때문에 CodeBase를 만드는 과정과는 별도로 진행한다. 최종적으로 Engine Execution 부분에서는 임의의 CodeBase를 선택한 뒤, 선택된 CodeBase에서 임의로 하나의 스테이트먼트를 선택한다. 이후 선택된 스테이트먼트는 available, liveness 정보를 가지고 있으므로 해당 정보를 이용해 변이에 알맞은 후보군을 CodeBrick 풀에서 찾는다. 이때, 후보군에 선정되는 스테이트먼트는 다음 조건을 만족해야 한다. 우선 첫 번째로 스테이트먼트가 사용할 변수 및 함수는 available 셋 안에 포함되어 있어야 하며, 두 번째로 liveness 셋이 있을 때는 반드시 살아있어야만 하는 변수를 변이시키기 이전 스테이트먼트와 똑같은 타입으로 선언해야 한다. 이러한 조건을 만족하는 CodeBrick 중 하나를 임의로 선택하여 기존 스테이트먼트를 CodeBrick으로 변경하여 코드를 변이시킨다. 최종

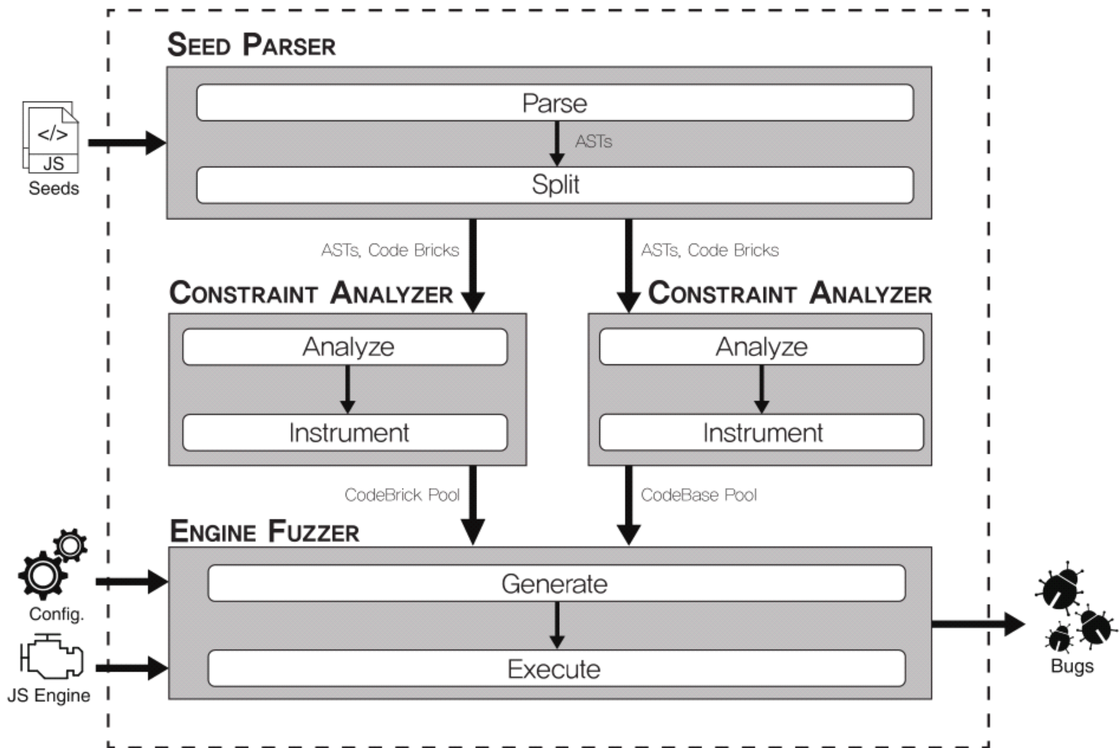

Fig. 3. Structure of the JMFuzzer

적으로 변이된 코드를 엔진을 통해 실행한다. 이와 같은 과정을 반복적으로 수행하고, 만들어진 변이된 시드 파일을 실행시켜 퍼징을 수행한다.

3.3 실행 예제

Fig.4. 는 JMFuzzer가 자바스크립트 코드를 변이하는 것을 나타낸 그림이다. 첫 번째 코드는 원본 코드에 해당하고, 두 번째 코드는 첫 번째 스테이트먼트가 변이된 상황을, 마지막 코드는 다섯 번째 스테이트먼트가 변이된 상황을 나타낸다.

첫 번째 코드를 보면, arr이란 변수가0 두 번째 스테이트먼트에서 사용되고 있으므로 live 한 변수임을 알 수 있다. 그뿐만 아니라 타입도 Array 타입이어야 함을 알 수 있다. 따라서 두 번째 코드에서

```
1 let arr = [1, 2, 3];
2 arr.length = 0x100000;
3 var num = 0x11;
4 arr.splice(0, num);
5 print (arr);
```

```
1 var arr = new Array(32);
2 arr.length = 0x100000;
3 var num = 0x11;
4 arr.splice(0, num);
5 print (arr);
```

```
1 let arr = [1, 2, 3];
2 arr.length = 0x100000;
3 var num = 0x11;
4 arr.splice(0, num);
5 delete(arr[num])
```

Fig. 4. Runtime example

다른 방식의 Array 타입을 가져오고, arr이라는 변수의 이름을 유지하며 변이한 것을 볼 수 있다.

세 번째 코드의 다섯 번째 스테이트먼트는 live 한 변수가 없고, arr이라는 Array 타입의 변수와 num이라는 number 타입의 변수가 available 하다. 따라서 해당 스테이트먼트를 arr와 num을 사용하는 스테이트먼트로 변이한 것을 볼 수 있다.

IV. 실 험

4.1 연구 질문

본 연구 실험에 관한 연구 질문은 다음과 같다.

질문1. 다양한 변이 코드를 생성함에 있어서 유효한 입력값을 생성할 수 있는가?

질문2. 퍼징 파라미터는 효과적인가?

질문3. State-of-the-art fuzzer들과 비교했을 때 많은 버그를 찾을 수 있는가?

질문4. JMFuzzer 는 실제 최신 자바스크립트 엔

진에서 버그를 찾을 수 있는가?

4.2 실험 환경

모든 실험은 88코어 Intel E5-2699 v4 (2.2 GHz) CPU와 512GB의 메인메모리를 가지고 있는 컴퓨터에서 진행했으며, Ubuntu 18.04 LTS 버전을 사용했다. Benchmark로 사용한 자바스크립트 엔진은 각각 메이저 브라우저 Edge, Chrome, Safari, FireFox에서 사용하는 자바스크립트 엔진을 사용했다. 4.3절부터 4.5절의 실험에서는 2018년 7월 기점의 엔진인 ChakraCore 1.10.1, V8 6.7.28.46, JavaScriptCore 2.20.3, SpiderMonkey 61.0.1 총 네 가지 버전의 엔진을 사용했으며, 4.6절의 실험에서는 2019년 10월 기점의 엔진인 ChakraCore 1.11.14, V8 7.7.299.13, JavaScriptCore 2.24.0, SpiderMonkey 70.0을 사용했다.

본 연구를 위해서는 오류가 발생하지 않도록 문법과 시맨틱의 올바름을 가진 자바스크립트 시드가 필수적이다. 따라서 이러한 조건에 부합하는 자바스크립트 시드를 찾기 위해 각 네 개의 자바스크립트 엔진의 regression test 셋을 각 오픈소스 사이트를 통해서 얻었고, 공식적인 ECMAScript standard[22]에 부합하는 테스트케이스를 가진 Test262[23]에서 수집했다.

4.3 변이된 자바스크립트 코드의 정확도

본 실험에서는 JMFuzzer가 올바른 문법과 시맨틱을 가진 자바스크립트 코드를 생성할 수 있는지를 실험한다. 이를 위해, 변이를 통해 생성한 자바스크립트 코드를 실행했을 때의 성공률을 측정하였다. Fig.5. 의 그래프에서 볼 수 있듯이, 코드를 변이시킬 때 시맨틱을 고려하는 경우가 시맨틱을 고려하지 않는 경우보다 높은 실행 성공률을 보였다. 이는 JMFuzzer의 동적 데이터 분석이 올바른 문법과 시맨틱을 갖는 시드를 생성하는 데 효과적임을 의미한다. 그래프의 x축은 주어진 코드에서 몇 개의 스테이트먼트를 변이했는지를 나타내며, 더 많은 스테이트먼트를 변이시킬수록 성공률이 낮아짐을 보여준다.

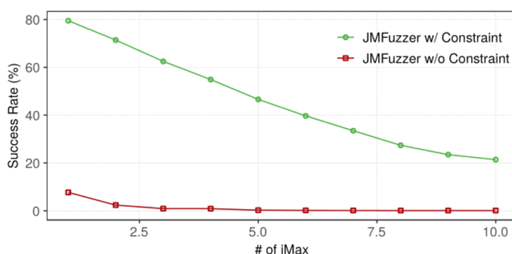

Fig. 5. Success rate of mutated JavaScript code

참고를 위하여, 시맨틱을 고려하는 기존의 자바스크립트 엔진 퍼저인 CodeAlchemist의 정확도와 비교하면 다음과 같다. CodeAlchemist 논문[7]에 따르면, CodeAlchemist가 생성하는 시드의 실행 성공률은 첫 번째 스테이트먼트를 실행했을 때 60.7%이며, 세 번째 스테이트먼트까지 실행했을 성공률은 30.0%까지 떨어진다. 반면 JMFuzzer는 1개의 스테이트먼트를 변이시켜 생성한 시드 중 79.5% 가 마지막 스테이트먼트까지 성공적으로 실행되며, 전반적으로 CodeAlchemist보다 더 유효한 시드를 생성하는 경향을 관찰할 수 있다. 한가지 주의할 것은, CodeAlchemist는 생성 기반(generational) 퍼저이므로, 주어진 유효한 시드를 변이시켜 새로운 코드를 생성하는 변이 기반(mutational) 퍼저인 JMFuzzer에 비해 낮은 실행 성공률을 보이는 것은 자연스러운 일이다. 대신, CodeAlchemist는 새로운 형태의 코드를 생성하는데는 JMFuzzer보다 유리하다. 따라서, 생성된 시드의 실행 성공률은 각 퍼저의 상이한 특성(생성 기반 혹은 변이 기반)을 고려하여 조심스럽게 해석되어야 한다.

4.4 파라미터 선택

Fig.6. 은 변경할 스테이트먼트의 개수(iMax)에 따라 발생한 버그 개수를 나타내는 그래프이다. 변이될 스테이트먼트의 개수는 각각 2, 4, 6, 8, 10개로

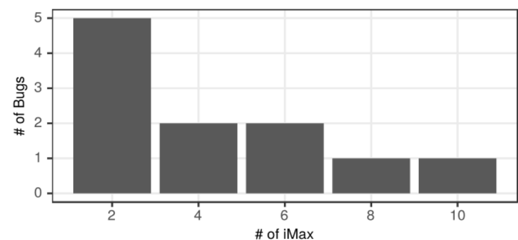

Fig. 6. Number of bugs according to number of statements

나누어 실험을 진행했다. 각각 24시간씩 총 3번에 걸쳐 실험하였으며, 결과의 중윗값을 표로 나타냈다.

실험 결과 스테이트먼트의 개수가 적을수록 많은 버그를 찾음을 확인할 수 있었으며, 개수가 많아질수록 적은 개수의 버그를 찾는 것을 확인할 수 있다. 특히 변이되는 스테이트먼트의 개수가 2개일 때 다른 결과와 비교해서 2.5배, 5배에 해당하는 버그를 찾았는데, 이는 스테이트먼트의 개수가 많아질수록 런타임 에러가 발생할 확률이 높아지며 코드를 생성하는 시간도 길어지기 때문이다. 이러한 이유로 이후에 진행되는 모든 실험에서는 기본 iMax 파라미터를 2로 설정하여 실험을 진행했다.

4.5 State-of-the-art 퍼저들 간의 비교

Table 2. 는 각각 자바스크립트 엔진에 대해 릴리즈, 디버그 모드 컴파일에서의 버그 개수를 정리한 표이다. JMFuzzer는 총 32개의 버그를, CodeAlchemist는 총 26개의 버그를, jsfunfuzz는 총 6개의 버그를, 마지막으로 Montage는 총 16개의 버그를 찾았다. JMFuzzer와 다른 state-of-the-art 퍼저와 비교했을 때 대부분 결과에서 JMFuzzer가 더 좋은 결과를 거둔 것을 볼 수 있다. 이때, V8과 SpiderMonkey에서는 버그가 나오지 않았는데, 이는 다른 최신 연구[7,8]에서도 공통으로 관찰되는 현상이다. 그 원인을 정확히 규명하기는 힘들지만, 해당 두 엔진이 다른 자바스크립트 엔진들보다 상대적으로 오랫동안 코드 관리가 잘 되어왔기 때문으로 추측해볼 수 있다.

Fig.7. 은 릴리즈 모드로 컴파일된 ChakraCore와 JavaScriptCore에서 발생한 중복된 버그를 비교한 그림이다. 왼쪽은 ChakraCore에서 얻어진 결과에 해당하며 오른쪽은 JavaScriptCore에서의 결과이다. 본 실험을 통해 JMFuzzer가 찾은 버그 일부는 다른 퍼저와도 공유하지만, 그렇지 않은 버그들도 많다는 것을 알 수 있다. 따라서 본 실험을 통해 JMFuzzer는 다른 state-of-the-art 퍼저와 다른 유형의 버그를 찾는 것을 확인할 수 있다.

Table 2. Compare JMFuzzer with state-of-the-art fuzzers with JavaScript engine

JS engine	JMF	CA	jff	Mont
Chakra(R, D)	5, 17	3, 14	0, 0	3, 6
JSC(R, D)	7, 3	4, 5	3, 3	3, 4
V8(R, D)	0, 0	0, 0	0, 0	0, 0
Spider(R, D)	0, 0	0, 0	0, 0	0, 0

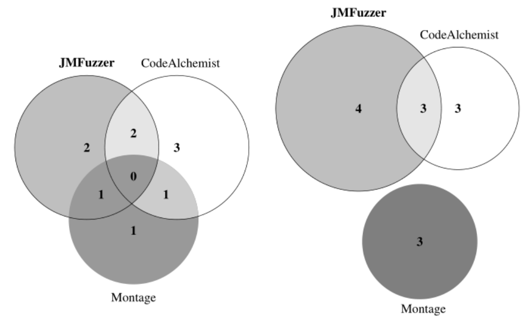

Fig. 7. Compare unique bug pool on JavaScript engine which is release compiled(ChakraCore, JavaScriptCore)

4.6 실제 환경에서의 버그 발견 사례

Table 3. 은 실제 찾은 버그들에 대한 설명을 간략히 해놓은 것이다. 각각의 버그는 대부분 메모리 커럽션 유형의 취약점이며, JIT 관련 버그를 포함하고 있다. ChakraCore와 V8 두 가지 엔진에서 총 5개의 버그를 찾았으며 이를 각각 회사에 보고한 상태이다.

Fig.8. 에서의 왼쪽의 코드는 버그의 시드 파일이고 오른쪽은 JMFuzzer가 찾은 버그를 간략히 해놓은 것이다. 16번째 라인에서 foo 라는 이름의 함수를 호출하게 되어있는데, JMFuzzer에서 available 한 foo 함수를 이용해 변이함에 따라 해당 코드가 생성되었으며, 이 테스트 코드는 함수의 jitting까지 고려하지 않았다. 따라서 본 버그 사례 예제를 통해 실제 발견되었던 버그를 변이를 통해 새로운 방향으로 트리거 할 수 있다는 것을 보였다.

위 실험들을 통해 4.1절에서 제시한 연구 질문에

Table 3. Unique bugs JMFuzzer found on latest JavaScript Engine

JS Engine	Impact	Status
Chakra 1.11.14	Likely Exploitable	Report
Chakra 1.11.14	Likely Exploitable	Report
Chakra 1.11.15	Not Exploitable	Report
Chakra 1.12.0	Not Exploitable	Report
V8 7.7.299.13	Likely Exploitable	Report

```
1  function bar() {
2    throw new Error();
3  }
4  function foo() {
5    try {
6      x = arguments;
7      bar();
8    } catch (e){
9      return x.length;
10   }
11   var x = {
12     j: 1,
13     k: 2.2
14   };
15 }
16 foo();
17 foo();
18 let pass = foo() === 0;
19
20 print(pass ? "Pass" : "Fail")
```

```
1  function boo(){
2    throw new Error();
3  }
4  function foo(){
5    try{
6      x = arguments;
7      bar();
8    } catch(e){
9      return x.length;
10   }
11   var x = {
12     j: 1,
13     k: 2.2
14   };
15 }
16 for(var v0 = 0; v0 < 0x1000; v0++){
17   foo();
18 }
19 let pass = foo() === 0;
20
21 print(pass ? "Pass" : "Fail")
```

Fig. 8. Bug case study

대한 답변을 얻을 수 있었다. 첫 번째로 4.3절의 실험을 통해 시맨틱을 유지하며 변이된 자바스크립트 코드와 무작위로 변이된 코드 간의 성공률 차이를 확인할 수 있었다. 모든 구간에서 10배 이상의 성공률 차이가 났으며, 이를 통해 시맨틱을 유지하는 방법이 유효한 입력값을 생성할 수 있음을 보였다. 두 번째로, 4.4절에서 진행한 실험을 통해 파라미터 값에 따라 버그의 개수가 2.5배에서 5배의 차이가 발생했으며, 이를 통해 퍼징 파라미터에 따른 효과를 확인할 수 있었다. 세 번째로 4.5절을 통해 state-of-the-art 퍼저들과 비교해서 JMFuzzer가 대체로 많은 버그를 찾는 것을 확인할 수 있었다. 마지막으로 4.6절에서 최신 자바스크립트 엔진에서 5개의 버그를 찾아, JMFuzzer가 최신 자바스크립트 엔진의 버그를 찾을 수 있음을 확인했다.

V. 결 론

본 연구에서는 동적 데이터 흐름 분석을 이용해 변이 방식을 자바스크립트 엔진 퍼징에 적용할 수 있는 방법을 도입한 JMFuzzer를 제안하고 유의미한 결과를 보였다. JMFuzzer는 동적 데이터 흐름 분석을 통해 변수들의 타입 정보 및 실행 순서를 알아내서 available, liveness 셋을 구해 변이기법을 적용하는 데 필요한 정보를 얻는다. 그리고 이 정보들을 이용해 다양한 변이 코드를 생성함에 있어서 유효한 입력값을 보임을 확인했으며, 퍼징 파라미터가 갖는 의미 또한 분석할 수 있었다. 그뿐만 아니라 state-of-the-art 퍼저들과 비교에서도 유의미한 결과를 보였다. 최종적으로 본 연구의 핵심 질문 중 하나였던 변이를 통해 생성한 유사한 자바스크립트 코드에서 버그를 찾을 수 있는가에 대한 답을 버그 사례를 통해 보였다. 최종적으로 JMFuzzer를 통해 4개의 메이저 브라우저에서 사용하고 있는 최신 자바스크립트 엔진에서 총 5개의 버그를 찾는 데 성공했고, 이를 각 회사에 전달했다. 마지막으로 제보한 버그 케이스가 regression test에 포함되는 등 본 연구를 통해 안전한 자바스크립트 엔진 사용 환경을 만드는 데 기여했다.

References

[1] Chromium, “ChakraCore Just-In-Time bugs,” https://bugs.chromium.org/p/project-zero/issues/list?q=chakra%20jit&can=1, Aug. 2019.

[2] Github, “CVE-2018-0834 Patch Commit,” https://github.com/Microsoft/ChakraCore/commit/6cd503299eac4a5b5ffc0c5bb0d072861f60e183, Aug. 2019.

[3] Github, “CVE-2018-0837 Patch Commit,” https://github.com/Microsoft/ChakraCore/commit/043257b7d47afab1240f5dd4cdd10bde38c574c3, Aug. 2019.

[4] Github, “CVE-2018-0840 Patch Commit,” https://github.com/Microsoft/ChakraCore/commit/24c7fa24623886859c31f9f1173e76977408fce2, Aug. 2019.

[5] Github, “Mozilla Security funfuzz,” https://github.com/MozillaSecurity/funfuzz, Jun. 2019

[6] Christian Holler, Kim Herzig, and Andreas Zeller, “Fuzzing with code fragments,” In Proceedings of the USENIX Security Symposium, pp 445-458, Aug. 2012.

[7] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha, “CodeAlchemist: Semantics-aware Code Generation to Find Vulnerabilities in JavaScript Engines,” In Proceedings of the Network and Distributed System Security Symposium, Feb. 2019.

[8] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son, “Montage: A

Neural Network Language Model-Guided JavaScript Fuzzer," In Proceedings of the USENIX Security Symposium, Aug, 2020.

[9] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo, "The art, science, and engineering of fuzzing: A survey," Apr. 2019.

[10] CERT, "Basic Fuzzing Framework," https://www.cert.org/vulnerability-analysis/tools/bff.cfm, Jun, 2019.

[11] Github, "radamsa," https://gitlab.com/akihe/radamsa, Jun. 2019.

[12] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury, "Directed greybox fuzzing," In Proceedings of the ACM Conference on Computer and Communications Security, pp 2329 - 2344, Nov. 2017.

[13] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen, "CollAFL: Path sensitive fuzzing" In Proceedings of the IEEE Symposium on Security and Privacy, pp 660 - 677, May. 2018.

[14] Github, "American Fuzzy Lop." https://github.com/google/afl, Jun. 2019.

[15] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in Proceedings of the IEEE Symposium on Security and Privacy, pp. 725 - 741, May. 2015.

[16] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in Proceedings of the USENIX Security Symposium, pp. 861 - 875, May. 2014.

[17] Samuel Groß, "FuzzIL: Coverage Guided Fuzzing for JavaScript Engines," Master Thesis, Karlsruhe Institute of Technology, Nov. 2018.

[18] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in Proceedings of the IEEE Symposium on Security and Privacy, pp. 579 - 594, May. 2017.

[19] J. Patra and M. Pradel, "Leaerning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data," TUDarmstadt, Tech. Rep. TUD-CS-2016-14664, 2016

[20] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "IFuzzer: An evolutionary interprter fuzzer using genetic programming," in Proceedings of the European Symposium on Research in Computer Security, pp. 581 - 601, Sep. 2016.

[21] Esprima, "Esprima," http://esprima.org, Jun. 2019.

[22] Ecma International. "ECMAScript 2015 language specification," https://www.ecma-international.org/ecma-262/6.0, Jun. 2015.

[23] Github, "Test262 ECMAScript conformance test suite," https://github.com/tc39/test262, Jun. 2019.

〈저 자 소 개〉

오 동 현 (Donghyeon Oh) 학생회원
2018년 2월: 단국대학교 소프트웨어학과 졸업
2018년 3월: 카이스트 정보보호대학원 석사 졸업
〈관심분야〉 소프트웨어 보안, 역공학, 바이너리 분석

최 재 승 (Jaeseung Choi) 학생회원
2015년 2월: 서울대학교 컴퓨터공학과 학사 졸업
2017년 2월: 서울대학교 컴퓨터공학과 석사 졸업
2017년 2월~현재: 카이스트 정보보호대학원 박사과정
〈관심분야〉 소프트웨어 보안, 소프트웨어 테스팅, 프로그램 분석

차 상 길 (Sang Kil Cha) 종신회원
2015년: 카네기멜론 박사 졸업
2015년~2020년: 카이스트 조교수
2020년~현재: 카이스트 부교수, 사이버보안연구센터장
〈관심분야〉 소프트웨어 보안, 자동화된 역공학, 바이너리 분석