Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.30, NO.3, Jun. 2020 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2020.30.3.443

최적화 컴파일된 자바스크립트 함수에 대한 최적화 해제 회피를 이용하는 런타임 가드 커버리지 유도 퍼저

김 홍 교,[1†] 문 종 섭[2‡]
[1,2]고려대학교 (대학원생, 교수)

Runtime-Guard Coverage Guided Fuzzer Avoiding Deoptimization for Optimized Javascript Functions

Hong-Kyo Kim,[1†] Jong-sub Moon[2‡]
[1,2]Korea University (Graduate student, Professor)

요 약

자바스크립트 엔진은 주로 웹 브라우저에 적재되어 웹 페이지를 표시하는 여러 기능 중 자바스크립트 코드를 입력으로 받아 처리하는 모듈이다. 자바스크립트 엔진 내 취약점은 종단 사용자의 시스템 보안을 위협할 수 있어 많은 퍼징 테스트 연구가 수행되었다. 그중 일부 연구는 자바스크립트 엔진 내 테스트 커버리지를 유도하는 방식으로 퍼징 효율을 높였으나, 최적화되어 동적으로 생성된 기계어 코드에 대한 커버리지 유도 퍼징은 시도되지 않았다. 최적화된 자바스크립트 코드는 예외적인 흐름 발생 시 코드를 해제하는 런타임 가드의 기능으로 인해 퍼징을 통한 충분한 반복 테스트가 어렵다. 본 논문은 이러한 문제점을 해결하기 위해 최적화 해제를 회피하여 최적화된 기계어 코드에 대해 퍼징 테스트를 수행하는 방법을 제안한다. 또한, 동적 바이너리 계측 방식으로 수행된 런타임 가드의 커버리지를 계측하고 커버리지 증가를 유도하는 방식을 제안한다. 실험을 통해, 본 연구가 제안하는 방식이 런타임 가드 커버리지, 시간당 테스트 횟수의 두가지 척도에서 기존의 방식보다 뛰어남을 보인다.

ABSTRACT

The JavaScript engine is a module that receives JavaScript code as input and processes it, among many functions that are loaded into web browsers and display web pages. Many fuzzing test studies have been conducted as vulnerabilities in JavaScript engines could threaten the system security of end-users running JavaScript through browsers. Some of them have increased fuzzing efficiency by guiding test coverage in JavaScript engines, but no coverage guided fuzzing of optimized, dynamically generated machine code was attempted. Optimized JavaScript codes are difficult to perform sufficient iterative testing through fuzzing due to the function of runtime guards to free the code in the event of exceptional control flow. To solve these problems, this paper proposes a method of performing fuzzing tests on optimized machine code by avoiding deoptimization. In addition, we propose a method to measure the coverage of runtime-guards by the dynamic binary instrumentation and to guide increment of runtime-guard coverage. In our experiment, our method has outperformed the existing method at two measures: runtime coverage and iteration by time.

Keywords: software testing, JavaScript engine, fuzzing, JIT compiler, coverage guidance

Received(03. 30. 2020), Modified(05. 13. 2020), Accepted(05. 14. 2020)

† 주저자, oxqo@korea.ac.kr
‡ 교신저자, jsmoon@korea.ac.kr(Corresponding author)

I. 서 론

자바스크립트 엔진은 주로 웹 브라우저에 적재되어 웹 페이지를 표시하는 여러 기능 중 자바스크립트 코드를 입력으로 받아 처리하는 모듈이다. 웹 브라우저에서 사용하는 자바스크립트는 신뢰할 수 없는 스크립트를 제한된 권한으로 안전하게 처리하도록 요구된다. 하지만 웹 브라우저 환경의 높은 성능 요구사항으로 인해 자바스크립트 엔진들의 프로그램 복잡도가 증가해 취약점 발생 확률을 높였다. 또한, 자바스크립트 엔진들은 성능 향상을 위해 C++과 같은 저수준 객체 지향 프로그래밍 언어로 구현되며, 많은 악의적인 공격들이 프로그램의 제어 흐름 탈취를 위해 저수준 프로그래밍 언어의 메모리 보호 실패를 이용한다[2]. 이러한 특징으로 인해 자바스크립트 엔진에 대해 많은 보안 연구가 이루어져 왔고, 퍼징 기법을 통해 주요 엔진에 존재하는 취약점을 사전에 탐지하여 자바스크립트 엔진의 보안 향상에 기여하는 많은 연구 역시 이루어졌다[18].

퍼징 기법은 소프트웨어에 대한 테스팅 기법의 하나로 무작위로 생성하거나 변조된 입력을 반복적으로 전달함으로써 소프트웨어에서 예외를 발생시켜 발생한 예외를 바탕으로 소프트웨어 내에 취약점을 발견하는 기법이다. AFL(american fuzzy lop)[4] 과 같은 발전된 기법을 사용하는 퍼저들은 소프트웨어 내에 다양한 경로에 대한 테스트를 위해 실행 흐름 계측에 의한 피드백을 사용하는 등 퍼징 테스트의 효율을 높이기 위한 기법들을 사용하기도 한다.

자바스크립트 엔진에 대한 기존의 퍼징 연구는 자바스크립트 언어의 문법을 준수하면서도 충분히 예외적인 행동을 유발하기 위한 입력을 만드는 연구와 자바스크립트 엔진 내 테스트 커버리지를 높이는 방식으로 진행되었다. Langfuzz[5]는 코드 조각을 이용한 자바스크립트의 변조 방식을 제안하였다. FuzzIL[1]은 유효한 문법의 자바스크립트를 생성하기 위해 중간 언어를 사용하는 퍼징 기법을 제안하였으며 자바스크립트 엔진 코드를 대상으로 커버리지를 증가시키는 방법을 제안하였다.

자바스크립트 엔진은 실행 과정에서 반복적으로 호출되는 스크립트의 수행 성능을 높이기 위해 JIT(just-in-time) 컴파일을 통해 컴파일하여 기계어 코드 형태로 메모리 내에 동적 생성한다. 컴파일 과정에서 최적화를 위해 피연산자의 자료형 등이 일정하게 반복된다면, 앞으로의 실행에서도 같은 형태가 반복될 것을 가정하여 컴파일한다. 이러한 가정은 생성된 기계어 코드 내에 삽입된 런타임 가드에 의해 보호되어 가정을 벗어나는 입력을 받으면 해당 예측을 통해 만들어진 기계어 코드를 더 이상 사용하지 않도록 하는 최적화 해제가 일어난다. 런타임 가드는 JIT 기계어 코드의 실행 흐름 보호를 위한 보안 검사이며 CVE-2019-5782[11]과 같은 많은 취약점이 이러한 보안 검사의 실패에서 기인한다.

예외적인 행동을 유도하기 위해 다양한 입력을 생성하는 퍼징 테스트의 특성으로 인해 JIT 컴파일로 최적화된 코드에 대한 퍼징은 최적화 해제를 빈번하게 일으키며 이는 해당 코드에 대한 퍼징 테스트의 효율을 저하시킨다. 생성된 기계어 코드가 클수록 코드 전체에 대한 테스트 수행 이전에 최적화 해제가 발생할 가능성이 크고 이로 인해 많은 부분이 테스트되지 않은 채로 해제된다.

본 논문은 이러한 문제점을 해결하기 위해 최적화 해제가 발생할 때 최적화 해제를 회피하여 최적화된 기계어 코드에 대해 퍼징 테스트를 수행하는 방법을 제안한다. 이 방법을 통해 한번 최적화된 코드에 대해서는 다양한 입력을 시도하면서도 최적화 해제가 발생하지 않은 채, 반복적인 퍼징 테스트가 가능하다. 또한, 비교적 큰 기계어 코드에 존재하는 런타임 가드의 보안 검사에 대해 통과 및 실패 여부를 측정하는 런타임 가드 기반 커버리지 측정 방식을 제안하고, 더 많은 보안 검사를 테스트하기 위해 런타임 가드 기반 커버리지를 증가시키는 방식의 퍼징 기법을 제안한다.

본 논문의 구성은 다음과 같다. 2장에서 기존의 자바스크립트 엔진 퍼징에 대한 연구를 기술한다. 3장에서 자바스크립트 엔진의 구조와 JIT 컴파일러의 특성, 4장에서는 본 논문에서 제안하는 퍼징 기법을 기술한다. 5장에서 제안한 기법을 적용한 테스트를 통해 성능을 평가하고 6장에서 결론과 향후 연구 방향을 제시한다.

II. 관련 연구

2.1 자바스크립트 엔진 퍼징

자바스크립트 엔진은 특정한 문법을 가지는 스크립트 언어를 입력으로 사용한다. 퍼징을 위해 변조되거나 생성된 입력은 자바스크립트의 문법을 준수하면서도 자바스크립트 엔진 내에서 예외적인 행동을 일

으켜야 한다. 이를 위해 자바스크립트 엔진을 대상으로 퍼징을 수행한 많은 연구는 문법에 부합하는 무작위 자바스크립트 코드를 생성하거나 기존의 자바스크립트 코드를 변조하면서도 문법 구조를 해치지 않는 방법에 집중해왔다.

Langfuzz[5]는 자바스크립트를 코드 조각으로 나누고 변조하는 방식으로 문법을 유지하면서도 예외적인 스크립트를 입력하는 방식을 사용한다. 제공된 샘플 자바스크립트 파일에 대해 코드 조각을 나누는 작업 이후 이들을 재조합하는 방식으로 새로운 스크립트를 생성한다.

Domato[9]와 같은 퍼저들은 사전에 정의된 문법들을 이용하여 새로운 자바스크립트 입력을 생성하는 방식을 채택한다. 이러한 방식은 사전에 문법을 정의하기 위해 많은 수작업이 필요하지만, 자바스크립트의 문법과 유사한 형태의 입력 스크립트를 생성할 수 있다. Domato는 추가로 에러를 방지하기 위해 각 라인을 try-catch 형태로 실행한다. try-catch 문으로 구성된 함수는 자바스크립트 엔진에서 JIT 컴파일러에 의해 의도와 다르게 해석될 여지가 있고[1], 일부 자바스크립트 엔진은 try-catch 블록이 포함된 함수에 대해 최적화를 지원하지 않으므로 최적화된 자바스크립트 함수에 대한 퍼징에 적절하지 않다.

Jsfunfuzz[8]는 퍼징을 위한 일부 기능에 자바스크립트를 사용한다. 자바스크립트가 정상적으로 작동하기 위해서 모든 자바스크립트 엔진들은 자바스크립트 내장 함수와 자료구조를 구현해야 한다. 이 구현의 메모리 내 표현은 자바스크립트 엔진 별로 다를 수 있으므로, 퍼저의 일부 기능을 자바스크립트로 구현하면 내장 함수 및 자료구조 명세를 엔진에 독립적으로 사용할 수 있어 여러 자바스크립트 기능 및 그 사용법을 정의하기에 용이하다.

2.2 커버리지 유도

퍼징 테스트는 대상 프로그램에 입력을 준 후 실행을 관찰하는 동적 분석 기법이므로 결과로 나타나는 프로그램 에러나 취약점은 비교적 쉽게 파악할 수 있으나, 전체 대상 프로그램에 대해 테스트의 완전성을 측정하는 것은 어렵다. 많은 퍼저들이 테스트의 완전성을 측정하는 척도로 커버리지라고 불리는 전체 대비 코드 수행률을 사용한다.

커버리지를 측정하는 방식은 측정 기준에 따라 나누어진다. 전체 함수 대비 실행된 함수를 나타내는 함수 커버리지(function coverage), 제어 흐름 그래프 내에서 분기 없이 반드시 순서대로 실행되는 코드 블록인 기본 블록(basic block)의 전체 대비 실행된 개수를 나타내는 엣지 커버리지(edge coverage), 전체 발생할 수 있는 모든 분기 대비 실행된 분기를 측정하는 분기 커버리지(branch coverage) 등이 있다.

커버리지를 측정하는 대표적인 계측(instrumentation) 방법으로는 소스코드에 커버리지 계측 구문을 추가하는 방식을 사용하는 정적 바이너리 재기록(static binary rewriting)방법이 있다. 소스코드가 존재하지 않는 작은 코드의 경우 런타임에 실행 코드를 삽입하는 동적 바이너리 계측(Dynamic Binary Instrumentation, DBI) 방법이 존재한다[13].

커버리지 유도를 사용하는 대표적인 퍼징 도구인 AFL의 경우 정적 바이너리 재기록과 에뮬레이터를 사용하여 동적 바이너리 계측을 수행하는 두 가지 방식을 모두 지원한다.

2.3 자바스크립트 엔진에 대한 커버리지 유도 퍼징

2008년 Godefroid[10]는 문법을 기반으로 하는 제약조건에 대한 기호 실행을 수행하는 퍼징 방법을 제시했다. 소스코드에 대한 분석을 통해 퍼징을 수행하는 화이트박스 퍼징에 적용한 결과, 인터넷 익스플로러7 브라우저에서 더 많은 경로를 탐색하고 문법 실패를 줄여 28%의 커버리지 향상을 보였다.

FuzzIL[1]은 2018년 공개되어 다수의 취약점을 찾은 커버리지 유도 자바스크립트 퍼저로 유효한 문법의 자바스크립트를 생성하기 위해 중간 언어를 사용하는 퍼징 기법을 제안하였으며 자바스크립트 엔진 코드를 대상으로 커버리지를 증가시키는 방법을 제안했다. 해당 논문의 결과물로 개발된 fuzzilli[6]는 자바스크립트 엔진 소스 코드에 clang의 sanitizer-coverage feature[19]를 적용해 정적 바이너리 재기록 방식으로 커버리지를 측정하기 때문에 JIT 컴파일러가 동적으로 생성한 코드에 대해서는 커버리지 유도를 지원하지 않는다.

III. 자바스크립트 엔진

3.1 자바스크립트 최적화

3.1.1 자바스크립트 JIT 컴파일

최근의 자바스크립트 엔진은 다층 실행 아키텍처를 사용한다[7]. 자바스크립트 함수는 파서에 의해 AST(Abstract Syntax Tree)로 변환되며 인터프리터에 의해 바이트 코드 형태로 실행된다. 바이트 코드에 의한 실행은 같은 기능을 자바스크립트 엔진이 실행되는 환경의 기계어 코드로 컴파일하여 실행하는 환경보다 느린 성능을 보인다. 하지만 컴파일을 통한 실행 역시 기능 수행 외에 컴파일 과정이라는 오버헤드가 존재하므로 자바스크립트 엔진의 JIT 컴파일러는 특정 바이트 코드가 빈번하게 실행된다고 판단했을 때 컴파일을 통해 기계어 코드를 생성한다[15].

3.1.2 최적화 과정

자바스크립트 엔진은 JIT 컴파일된 코드를 더욱 단순화하여 실행 성능을 향상하기 위해 추가적인 JIT 컴파일러를 활용할 수 있으며 실행 도중 수집되는 정보의 프로파일링(profiling)을 통한 가정을 활용한다. 예를 들어 자바스크립트의 '+' 연산은 숫자 자료형 사이의 덧셈과 문자열 사이의 병합을 모두 나타낼 수 있다. 특정 함수의 실행 과정에서 '+'연산이 숫자 자료형 사이의 덧셈으로만 사용되었다면 자바스크립트 엔진은 해당 연산이 숫자 자료형 사이의 덧셈이라고 가정하여 해당 연산을 숫자를 더하는 기능만 수행하도록 최적화한다. 자료형뿐만 아니라 숫자의 오버플로우 여부, 접근 가능한 메모리 영역, 메모리에 나타나는 변수의 구조 등 다양한 형태의 가정이 존재할 수 있다. 자바스크립트가 자바스크립트 엔진에 의해 컴파일되는 일반적인 구조는 Fig.1.과 같다.

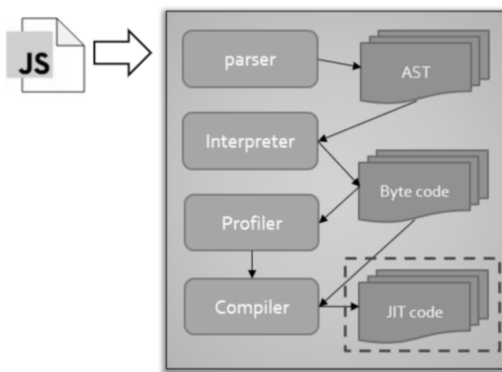

Fig. 1. Process of JIT Compilation

3.1.3 런타임 가드와 최적화 해제

최적화 과정에서 세운 가정이 지켜지지 않아 다른 유형의 실행 흐름이 발생한 경우 자료 구조에 대한 착오(type confusion), 메모리 유출(memory leak), 해제 후 사용(use-after-free) 메모리에 대한 임의적인 수정(out-of-boundary write) 등의 취약점이 발생할 수 있다. 이를 방지하기 위해 최적화 컴파일 중 세워진 가정이 이후의 실행에서 위반되지 검사하는 런타임 가드가 기계어 코드 내에 추가되고, 위반 발생 시 최적화된 코드가 아닌 기존의 실행 방식으로 돌아간다. 이 과정을 최적화 해제라고 한다[14]. 최적화 해제된 기계어 코드는 캐시를 통해 다음 최적화가 일어날 때 재사용될 수 있다.

3.2 최적화된 기계어 코드에 대한 퍼징

3.2.1 개발자용 셸을 이용한 퍼징 수행

자바스크립트 엔진은 주로 웹 브라우저에 내장되어 자바스크립트를 수행하지만, 브라우저는 HTML, CSS, 미디어 파일 등을 표시하기 위한 기능과 이를 렌더링하여 사용자에게 표시하는 기능, 네트워크 기능 등 핵심적인 자바스크립트 기능 이외의 많은 기능을 내포한다. 이 때문에 웹 브라우저를 직접 퍼징하는 방식은 주로 사용되지 않는다. 대부분의 자바스크립트 엔진은 로컬 커맨드 라인 인터페이스 환경에서 엔진을 테스트하기 위한 개발자용 셸을 제공한다.

개발자용 셸은 자바스크립트 파일을 적재하여 실행하는 기능과 표준 입력으로부터 유동적으로 입력되는 자바스크립트 라인을 실행하는 기능을 가진다. 표준 입력으로 전달된 라인을 실행하는 기능을 활용하면 자바스크립트 엔진의 실행을 관찰한 후 실행할 다음 입력을 결정할 수 있으므로 본 연구에서는 Fig.2.와 같이 자바스크립트 엔진을 구동하는 개발자용 셸에 대해 퍼징 테스트를 수행하는 시스템 구성을 사용한다.

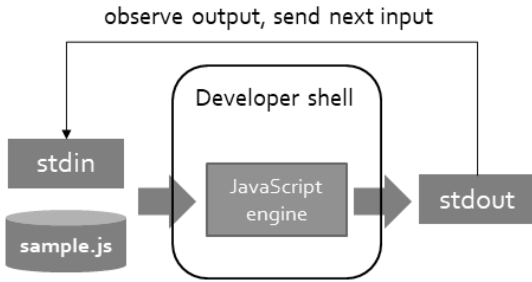

Fig. 2. Structure of JavaScript Engine Test Using Developer Shell

3.2.2 최적화된 기계어 코드에 대한 퍼징 방법

최적화 컴파일된 코드에 대한 퍼징을 지원하는 퍼저들은 Fig.3.과 같이 준비된 입력 자바스크립트의 최적화를 유도한 뒤 가정 외의 인자를 해당 함수에 전달하는 방식으로 퍼징 테스트를 수행한다. 일부 자바스크립트 엔진은 단순하고 작은 함수에 한해 즉각적인 최적화를 수행하기도 하지만, 자바스크립트 엔진이 최적화를 결정하게 하는 일반적인 방법은 같은 인자를 이용한 반복적인 호출이다[16]. 반복호출을 이용해 대상 함수의 최적화가 발생하면 컴파일된 기계어 코드 내에서 예외적인 실행 흐름을 발생시키기 위해 변조되거나 새로 생성된 인자를 사용하여 해당 함수를 호출한다. Fig.3.에서 'DEOPT'는 최적화 해제를 의미한다.

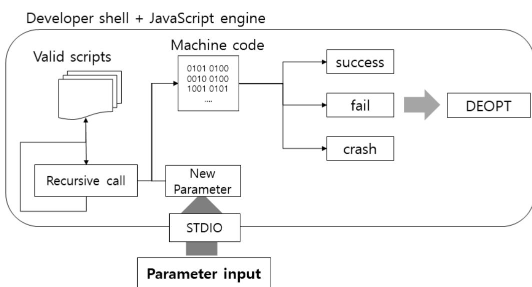

Fig. 3. Process of Fuzzing against Optimized JavaScript Code

3.2.3 최적화 해제로 인한 퍼징 오버헤드

최적화된 함수를 변조된 인자와 함께 호출하여 변조된 인자가 자바스크립트 엔진의 가정 실패를 유도한 경우 런타임 가드에 의해 최적화 해제가 발생하고, 다음 호출 시 해당 함수는 최적화된 기계어 코드를 통해 실행되지 않는다. 함수를 최적화된 상태에서 다시 테스트하기 위해서는 다시 반복호출을 통해 자바스크립트 엔진이 최적화를 결정하도록 유도해야 한다. 이 과정에서 입력 인자가 충분히 예외적일수록 더 많은 최적화 해제가 일어나며 최적화 유도를 위한 오버헤드가 더 커지게 된다.

일부 자바스크립트 엔진은 디버깅 환경에 한해 사용 가능한 특수한 문법을 이용하여 수동으로 특정 함수에 대해 최적화를 발생시키는 기능을 제공한다. 하지만 최적화를 위해서는 가정을 위한 충분한 실행 경험이 누적되어야 하며 반복호출 없이 수동으로 생성된 기계어 코드는 실제 반복호출을 통해 생성된 기계어 코드와 현저히 다를 수 있다. 따라서 오탐(false positive)을 줄이기 위해서는 반복적인 호출을 통한 최적화 유도가 필수적이다.

IV. 최적화 해제 회피 및 런타임 커버리지 유도 방법론

런타임 가드는 최적화 과정에서 세워진 가정을 보호하기 위한 보안 제약으로써, 최적화된 기계어 코드에 대한 퍼징은 이러한 제약의 미비로 인한 오류를 찾는 것을 목적으로 한다. 본 논문에서는 3.2 최적화된 기계어 코드에 대한 퍼징에서 언급한 기존 방식의 문제점을 해결하기 위해 퍼징 시스템 내에 디버거와 실행 흐름 계측을 위한 기계어 코드를 추가하는 방식을 제안한다. 또한, 최적화된 기계어 코드의 자체 보안 검사인 런타임 가드에 대해 선택적으로 동적 바이너리 계측을 적용하는 런타임 가드 기반 커버리지 분석 방식을 제안한다. 이 방법을 통해 최적화 해제로 인한 오버헤드를 줄여 시간 대비 검사의 효율성 증대와 더 많은 런타임 가드에 의한 보안 검사를 테스트하도록 퍼저를 유도할 수 있다.

4.1 제안 방법의 전체 구성

퍼징 시스템의 구성은 Fig.4.와 같다. 최적화 해제를 회피하며 런타임 가드 기반의 커버리지 증가를 유도하기 위해 Fig.3.에 공개된 기존의 방식에 Fig.4.에 검은색으로 표시된 구성 요소들을 추가한다. 디버거 모듈, 커버리지 계측 모듈, 커버리지 유도 모듈, 그리고 인자 변조 모듈이 추가된다.

디버거 모듈은 최적화 해제 회피를 위해 최적화 해제 루틴에 브레이크포인트를 삽입하고 최적화 해제

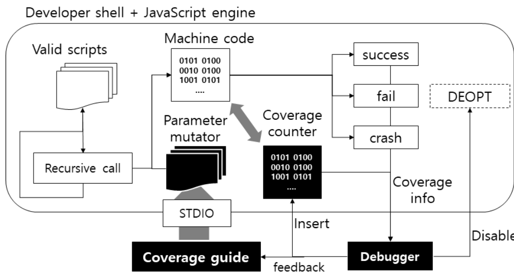

Fig. 4. Suggesting Process of Fuzzing against Optimized JavaScript Code

발생 시 자바스크립트 엔진이 해제 루틴을 건너뛰게 한다. 또한, 최적화된 기계어 코드에 대한 동적 바이너리 계측을 위해 자바스크립트 엔진 메모리 내에 기계어로 된 커버리지 계측 도구(Coverage counter)를 삽입한 뒤 런타임 가드에 의해 계측 도구가 호출되게 한다. 커버리지 계측 도구 내 존재하는 커버리지 정보(Coverage info)는 최적화된 함수 종료 시 디버거 모듈이 대상 엔진의 메모리를 읽어 수집한다.

커버리지 증가 유도 도구(Coverage guide)는 디버거 모듈에 의해 수집된 커버리지 정보를 바탕으로 다음 호출에 전달될 변조된 인자 생성에 관여한다. 측정된 커버리지의 분석 결과를 표준 입출력을 통해 인자 변조 모듈(Parameter mutator)에 전달하여 인자 변조 모듈이 다음에 변조할 인자를 지정한다.

변조된 인자를 생성하고 이를 통해 최적화된 기계어 코드를 실행하기 위해서는 자바스크립트 형태로 구현된 인자 변조 모듈을 사용한다. 자바스크립트를 사용하여 인자 변조 부분을 구현하여 자바스크립트 엔진에 독립적으로 변조된 인자를 생성할 수 있게 한다.

4.2 디버거 모듈 기능 구성

디버거 모듈은 본 연구의 퍼징 시스템에서 분석의 대상이 되는 자바스크립트 엔진의 실행 흐름을 파악하고, 필요에 따라 실행 흐름을 제어하는 역할을 한다.

최적화를 통해 분석할 자바스크립트 함수가 정해지면 반복적인 호출을 통해 대상 함수의 최적화를 유도해야 한다. 기존 방법의 경우 높은 횟수로 반복호출을 수행한 후 대상 함수가 최적화되었다고 가정한 상태로 변조된 인자를 전달한다. 디버거 모듈이 추가되면 적은 횟수의 반복 후 자바스크립트 엔진이 대상 함수를 최적화하는지를 파악할 수 있어, 함수가 최적화된 이후에 호출을 반복하는 오버헤드나 최적화되지 않은 함수에 변조된 인자를 전달하는 테스트 실패를 예방할 수 있다. 디버그 모듈은 자바스크립트 엔진의 최적화 기능 내에 브레이크포인트를 삽입해 대상 함수가 최적화되는 시점을 파악하고 컴파일된 기계어 코드의 메모리상 위치와 런타임 가드의 위치를 확인한다.

변조된 인자로 인해 대상 함수의 최적화가 해제되면 최적화를 재시도하기 위한 반복 오버헤드가 발생한다. 디버거 모듈은 자바스크립트 엔진의 최적화 해제 함수에 브레이크포인트를 삽입해 대상 함수에 대한 최적화 해제가 시도될 때 이를 무력화하여 반복 오버헤드를 줄인다.

또한, 디버거 모듈은 대상 엔진의 메모리에 새로운 페이지를 삽입하여 실행 흐름 계측을 위한 코드와 계측 결과를 저장할 메모리를 확보한다. 이후 최적화된 기계어 코드 내 런타임 가드 분기가 계측 코드 영역을 호출하도록 코드를 수정한다. 대상 함수가 종료되면 계측 결과가 저장된 메모리를 읽어 계측 결과를 확인하고 변조할 인자를 선정하는 과정에 적용하여 새로운 경로 탐색을 유도한다.

4.3 런타임 가드 기반 커버리지 측정

본 연구에서는 최적화된 기계어 코드의 런타임 가드 기반의 분기 커버리지를 선택적으로 측정하기 위해 자체 개발한 동적 바이너리 계측 모듈을 사용한다. 최적화를 통해 생성된 기계어 코드 내에서 디버거 모듈에 의해 파악된 최적화 해제 루틴으로 분기하는 코드를 파악한다. 해당 분기 코드들은 분기 명령어별로 구성된 계측 코드를 호출하는 명령어로 대체된다.

계측 코드 내에서는 수정 전의 원본 분기 명령어를 실행하여 정상적인 실행 흐름에서 분기의 발생 여부를 판단한다. 분기 발생 여부에 따라 각각의 상황에 해당하는 계측 결과 메모리의 값을 증가시켜 실행 과정에서 해당 분기가 몇 번 발생했는지를 측정한다. 예를 들어 Fig.5.에 나타나는 런타임 가드가 최적화 해제로 분기하지 않는 β_1과 가정 실패로 최적화 해제를 일으키는 β_2 분기에 대해 각 분기의 실행 횟수

를 메모리 내에 기록한다. 기록 이후 수행된 분기에 맞는 코드 위치로 이동하기 위해 스택 내의 실행 관련 정보를 동적으로 수정하는 기법인 OSR(on-stack replacement)[17] 기법을 사용

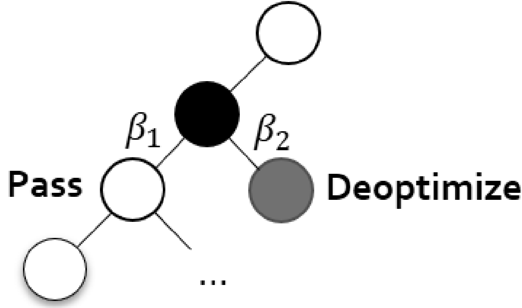

Fig. 5. Runtime-Guard Branches

한다. 또한, 기존의 실행 흐름에 영향을 미치지 않기 위해 모든 범용 레지스터들을 새로 매핑된 메모리에 저장하며, 계측 종료 시 저장한 레지스터를 복원한다. 이 과정은 Table 1.의 의사 코드와 같이 수행된다.

Table 1. Pseudo Code for Dynamic Binary Instrumentation example

```
runtime_guard:
cmp rax, 0x3
jne deoptimize → call branch_counter_jne

normal_path:
...

barch_counter_jne:
(save_registers)
jne taken
jmp not_taken

taken:
(increase taken branch counter)
(OSR, return address → deoptimize)
jmp eplilogue

not_taken:
(increase not taken branch counter)
(OSR, return address → normal_path)

epilogue:
(load_registers)
ret
```

4.4 런타임 가드 기반 커버리지 증가 유도

자바스크립트 함수가 이미 기계어 코드로 최적화되었다면, Fig.4.와 같이 함수의 수행 결과는 정상적인 반환 코드가 수행되는 경우와 에러가 발생하는 경우, 런타임 가드로 인한 최적화 해제가 발생하는 경우로 나누어진다. 디버거 모듈은 각각의 종료 결과를 파악하고 함수의 종료 시 런타임 가드로 인한 분기들의 실행 횟수를 읽는다. 한 번의 실행에서 모든 런타임 가드 분기별 실행 횟수에 대한 집합을 하나의 경로로 취급한다.

퍼저는 이렇게 발생한 경로가 새로운 경로일 경우 자바스크립트 엔진에 명령을 전달하여 인자 변조 모듈 내 자바스크립트 배열 형태의 인자 풀 변수에 해당 경로를 발생시킨 인자를 저장한다. 중복된 경로가 발생한 경우 해당 경로의 실행 횟수를 증가시킨다.

최적화된 함수에 대한 다음 호출에서 인자로 사용될 변조된 인자를 생성하기 위해 입력 풀에서 하나의 인자 집합을 선정한다. 해당 인자의 집합으로 실행된 경로가 여러 번 나타날수록 해당 인자 집합이 선정될 확률을 낮춰 새로운 경로를 탐색한 인자 집합이 선정될 확률을 높인다. 선정된 인자 집합을 인자 변조 모듈에 전달하는 방식을 통해 전체적인 탐색 경로를 증가시키는 방향으로 인자의 변조를 제어한다. 인자 집합을 선정하기 위한 확률의 집합 W를 생성하는 과정은 (1)과 같다.

$$\begin{aligned} P &= \{(\pi_1, c_1), (\pi_2, c_2), ..., (\pi_N, c_N)\} \\ P' &= \left\{(\pi_i, c_i') \mid \forall (\pi_i, c_i) \in P . c_i' = \frac{1}{c_i}\right\} \\ S' &= \sum_{i=1}^{N} c_i' \\ W &= \left\{w_i \mid \forall (\pi_i, c_i') \in P' . w_i = \frac{c_i'}{S'}\right\} \end{aligned} \quad (1)$$

중복되지 않는 경로 π와 해당 경로가 실행된 횟수 c의 쌍을 가지는 집합 P에서 c_i와 반비례하는 가중치 c'^{i}를 가지는 집합 P'를 도출한다. P'에서 가중치 c_i'를 모든 가중치의 총합인 S'으로 나눈 값이 π_i 경로를 발생시키는 인자가 풀에서 선택될 확률이 된다.

새로운 경로를 발생시킨 인자를 자바스크립트 내부에 저장하거나 실행 경로를 증가시키기 위해 변조

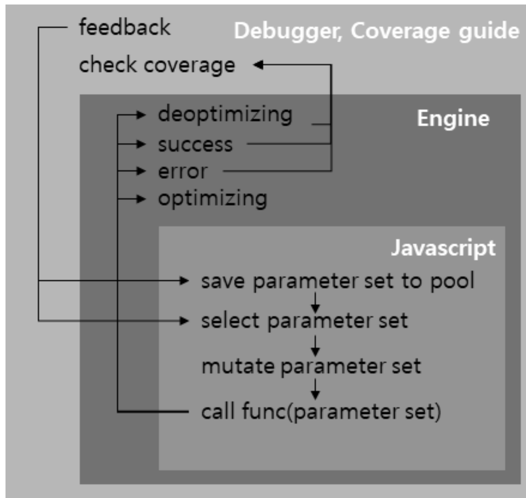

Fig. 6. Runtime guard based coverage guidance architecture

될 인자 집합을 결정하면 입출력 도구를 통해 자바스크립트 엔진에 명령이 전달된다. 경로 확인 및 명령 전달의 전 과정은 Fig.6.과 같다.

V. 실험 및 평가

5.1 실험 환경

실험은 Table 2.의 하드웨어 성능을 가진 가상 환경과 대상 소프트웨어에 대해 수행되었다. 퍼징 대상 자바스크립트 엔진은 구글사의 크롬 브라우저에 사용되는 v8엔진[12] 소스코드를 빌드한 결과물을 사용했다. 5.4 제안 방법의 성능 측정 실험에는 Target 1, 5.5 기존 취약점 재연 실험에는 Target 2에 표시된 버전의 v8을 사용했다. 빌드 옵션은 디버그를 나타내는 debug, 기계어의 아키텍처를 나타내는 x64 옵션을 사용했고 나머지 옵션은 기본 제공 옵션을 사용했다. v8엔진을 로드하여 테스트할 개발자용 셸로 v8의 소스코드 빌드 시 함께 생성되는 d8 실행파일을 사용했다.

Table 2. Experiment environment

System	
OS	Ubuntu 18.04.4 LTS x86_64
kernel	Linux 5.3.0-42-generic
CPU	Intel(R) Core(TM) i5-6400
RAM	8GB
Target 1	
name	V8
version	version 7.8.0 (candidate)
commit	f584f7cc1b54cd502bc159136bb599dccc05ef7b
architecture	x64
build	debug
Target 2	
name	V8
version	version 7.4.0 (candidate)
commit	c22bb466d8934685d897708119543d099b9d2a9a
architecture	x64
build	debug

5.2 최적화된 자바스크립트 함수에 대한 퍼징

5.2.1 최적화할 자바스크립트 함수 샘플 생성

최적화할 자바스크립트 함수를 생성하기 위해서는 fuzzilli[6] 퍼저의 소스코드를 수정하여 퍼저가 생성한 입력 중 자바스크립트 엔진에서 문법 오류 없이 정상적으로 실행된 함수 900개를 사용하였다.

5.2.2 자바스크립트 함수 수정

최적화를 통해 기계어 코드로 컴파일된 함수는 호출인자를 입력으로 받는다. 호출인자가 존재하지 않으면 함수는 매번 같은 경로를 수행하게 되고, 입력의 변조로 인해 예외적인 실행 흐름을 만들어 낼 수 없다.

본 연구는 인자를 통해 기계어 코드로 컴파일된 자바스크립트 함수 내에서 예외적인 실행 경로를 만들어 낼 수 있도록, 함수 내에 존재하는 무작위 내부 변수를 호출 인자로 대체하고 함수의 선언 과정에 호출 인자를 추가하는 함수 변조 모듈을 구현했다. 수집된 자바스크립트 함수는 함수 변조 모듈에 의해 호출인자를 내부에서 사용하는 방식으로 수정되며, 같은 자바스크립트 함수 샘플에 대해서도 변조 모듈이 변조한 내부 변수에 의해 매번 다른 형태를 가지게 된다.

5.3 성능 평가 척도

본 연구가 제안하는 방법의 목적은 첫째, 최적화 해제로 인한 오버헤드를 줄여 시간 대비 많은 테스트를 수행하는 것과 둘째, 기존의 방식으로 도달하지 못한 경로에 있는 런타임 가드에 대한 테스트를 수행

Table 3. Experiment result

	Configure 1	Configure 2	Configure 3
Iterations	214820.0	200231.67	64500.33
Covered path	497.67	485.33	418.33
Total runtime-guards	144292975.33	141104216.67	54561603.0
Avg. iter. for repro.	16029.59	-	283513.83

하는 것이다. 구현 결과가 상기 목적을 달성하는지 확인하기 위해 반복 횟수, 발견한 경로의 개수, 수행된 총 런타임 가드 분기의 개수를 평가 척도로 사용한다.

반복 횟수는 주어진 시간 동안 얼마나 많은 테스트를 수행했는지 보여주는 지표이다. 발견한 경로의 개수는 퍼징을 수행하며 해당 방법으로 발견한 경로의 총합이며, 각각의 방법이 얼마나 많은 새로운 경로를 탐색했는지 보여준다. 수행된 총 런타임 가드 분기의 개수는 보안 장치인 런타임 가드의 보호 실패를 찾기 위해 얼마나 많은 런타임 가드가 실행되었는지 보호 기능이 검사된 횟수를 보여준다. Avg. iter. for repro.는 5.5 기존 취약점 재연 실험에서 크래시 발생 시점까지의 평균 반복 횟수를 나타낸 항목이다.

5.4 제안 방법의 성능 측정

5.4.1 실험 구성

실험의 구성은 본 논문에서 제안하는 방법에 의한 성능 평가 척도의 변화를 나타내기 위해 최적화 해제 회피, 런타임 가드 커버리지 유도의 두가지 제안 방법에 대한 적용 여부를 기준으로 Configure 1, Configure 2, Configure 3으로 나누어 진행했다. Configure 1은 최적화 해제 회피와 런타임 가드에 대한 커버리지 증가 유도를 모두 사용한 실험이며, Configure 2는 커버리지 유도 없이 최적화 해제 회피만을 사용한 실험, 그리고 Configure 3은 적용된 기법 없이 기존의 방법을 재연한 실험이다.

동일한 조건에서의 실험을 위해 Fig.4.를 구현한 퍼저에서 최적화 해제 회피와 런타임 가드 커버리지 유도 기능을 인자를 통해 조절할 수 있도록 했다. 기존의 방법을 재연한 Configure 3은 기존 퍼저로 대체할 수 있으나, 입력 변조 기능의 성능 및 구현 언어와 실행 환경의 차이로 제안하는 기법 적용에 의한 차이를 정확히 파악하기 어려우므로 구현된 퍼저 프로그램에서 제안하는 두 가지 기법을 제거한 상태로 실험했다.

성능 평가 척도에 대해 시간 대비 효율을 보이기 위해 1회 실험에서 각각의 Configure가 300개의 샘플 자바스크립트 함수를 함수별로 같은 시간 동안 테스트했으며 다른 샘플에 대해 해당 실험을 총 3회 반복하여 측정된 평가 척도의 평균을 Table 3.과 같이 측정했다. 함수당 퍼징 시간은 대상 자바스크립트 함수가 최적화되기까지의 시간과 최적화 이후 충분한 반복 횟수 동안 경로를 탐색하기까지의 시간을 부여하기 위해, 실험이 진행된 시스템 내에서 기존 방법인 Configure 3을 기준으로 평균 150회 이상의 반복이 가능한 시간인 2분으로 지정했다.

5.4.2 실험 결과 및 해석

커버리지 유도와 최적화 해제 무력화를 모두 적용한 실험은 어떠한 방법도 적용하지 않은 실험 대비 2.3배 많은 반복(iteration)을 수행했다. 이를 통해 최적화 해제로 인한 오버헤드가 테스트 효율에 결정적인 영향을 미치는 것을 알 수 있다. 커버리지 유도를 적용한 실험이 적용하지 않은 실험에 비해 7% 더 많은 반복을 수행했다. 커버리지 유도를 위한 별도의 오버헤드가 있지만, 최적화 해제가 일어나는 경로를 더 많이 찾았기 때문에 짧은 경로를 실행하여 더 많은 반복을 수행한 것으로 추측된다.

커버리지 유도를 사용하지 않은 두 개의 실험 중 최적화 해제를 무력화한 실험이 그렇지 않은 실험에 비해 평균 약 67개의 더 많은 경로를 찾았다. 반복 횟수의 차이가 경로 탐색에도 많은 영향을 미치는 것을 알 수 있다. 커버리지 유도와 최적화 해제 무력화를 모두 사용한 실험은 최적화 해제 무력화만을 사용한 실험에서 찾지 못한 평균 12.3개의 추가적인 경로를 찾아 커버리지 유도가 효과적임을 보였다. 커버리지 유도를 사용한 실험은 사용하지 않은 실험에 비

해 같은 경로를 탐색한 수의 차이가 작게 나타났기 때문에, 통제된 실험 환경이 아니라 일정 반복 이상 새로운 경로가 발견되지 않으면 다음 샘플을 선택하는 일반적인 환경에서는 더 많은 경로를 찾을 수 있을 것으로 보인다. 제안하는 방법을 모두 적용한 실험은 기존의 방법 대비 약 19% 많은 새로운 경로를 탐색했다. 비교적 작은 코드인 최적화된 기계어 코드에만 제한적으로 적용된 것을 고려하면 의미 있는 증가로 보인다.

테스트한 총 런타임 가드 분기의 개수는 커버리지 유도와 최적화 해제 무력화를 모두 사용한 실험에서 기존의 방법 대비 164% 많은 테스트가 수행되었다. 커버리지 가이드를 사용한 실험은 사용하지 않은 실험에 비해 2% 많은 런타임 가드 분기를 수행했다.

5.5 기존 취약점 재연 실험

5.5.1 실험 구성

기존 취약점의 재연 실험을 통해 취약점 탐지의 효율성을 보이기 위해 개념 증명 코드가 존재하는 취약점을 포함하는 버전의 v8 엔진을 대상으로 실험을 수행했다. 대상 엔진은 Table 2.의 Target 2와 같다. 취약점 재연을 위해 공개된 개념 증명 코드[20]를 사용했다. 해당 개념 증명 코드는 취약점을 발생시키는 최소한의 코드만 남겨놓은 코드이며 실제 퍼징 과정에서 생성되는 함수는 취약점 발생에 필요한 부분 이외에 많은 라인이 존재하므로 5.4 제안 방법의 성능 측정에 사용한 함수 샘플 내부에 개념 증명 코드의 내용을 삽입하여 퍼징 대상 함수를 생성했다.

실험은 5.4 제안 방법의 성능 측정에서 사용한 Configure 1과 Configure 3이 샘플 함수 내에 존재하는 개념 증명 코드 부분에서 크래시를 발생시키는 시점까지 수행한 반복 횟수를 측정하도록 구성했다. 시간의 제약을 두지 않기 때문에 최적화 해제 회피만 적용한 설정인 Configure 2는 Configure 3과의 유의미한 차이가 없을 것으로 판단되어 Configure 2에 대한 실험은 수행하지 않았다.

5.5.2 실험 결과 및 해석

Table 3.의 Avg. iter. for repro.는 크래시 발생 시점까지의 평균 반복 횟수를 나타낸 항목이다. 인자 변조 기능의 무작위성으로 인한 실험 결과의 편차를 줄이기 위해, 각각의 Configure에서 100번의 크래시를 발생시킨 후 반복 횟수의 평균값을 소수점 둘째 자리까지 표시했다. 실험 결과 크래시를 발생시키기까지 Configure 1은 평균 16029.59회의 반복이 필요했으며 Configure 3은 평균 283513.83회의 반복이 필요하여 기존 방법이 본 논문에서 제안하는 방법 대비 약 17.69배의 반복을 수행해야 기존의 취약점을 재연할 수 있었다.

실제 퍼징 테스트는 다양한 함수를 대상으로 테스트하기 위해 하나의 함수에 대한 반복 횟수를 제한한다. 따라서 본 논문에서 제안하는 방법을 적용한 Configure 1이 크래시까지 도달하기 위해 많은 반복을 필요로 하는 Configure 3보다 더 높은 취약점 탐지 성능을 보이는 것으로 해석된다.

VI. 결 론

본 논문에서는 런타임 가드로 인한 최적화 해제가 최적화된 자바스크립트 함수에 대한 퍼징 효율을 저해하는 것에 착안하여 최적화 해제를 회피하여 최적화된 자바스크립트 함수를 퍼징하는 방법을 제안했다. 또한, 최적화된 자바스크립트 내의 런타임 가드의 분기를 계측하여 더 많은 보안 검사에 대한 테스트가 가능하도록 유도하는 퍼징 방식을 제안했다.

실험을 통해 최적화 해제 회피가 최적화된 자바스크립트 함수에 대한 퍼징 효율을 증가시키는 것을 보였으며 커버리지 유도를 통해 기존의 방식에서 수행되지 않았던 런타임 가드의 분기를 테스트할 수 있음을 보였다.

향후 런타임 가드 기반 커버리지의 증가를 위해 변조할 호출 인자를 선정하는 과정 외에도 호출 인자를 변조하는 방식에 있어서 수행 경로 정보를 활용하면 더 많은 경로를 빠르게 탐색할 수 있을 것으로 예상된다.

References

[1] S. GROß, "FuzzIL: Coverage Guided Fuzzing for JavaScript Engines," Ph.D. Thesis, Karlsruhe Institute of Technology, Jan. 2018.

[2] D. Jang, Z. Tatlock, and S. Lerner, "SafeDispatch: Securing C++ Virtual Calls from Memory Corruption

Attacks," NDSS Symposium 2014, Feb. 2014.
〔3〕 G. A. Perez, C. M. Kao, Y. C. Chung, and W. C. Hsu, "A hybrid just-in-time compiler for android: comparing JIT types and the result of cooperation," Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embed-ded systems, pp. 41-51, Oct. 2012.
〔4〕 Lcamtuf, "american fuzzy lop" http://lcamtuf.coredump.cx/afl/, Mar. 18, 2020
〔5〕 C. Holler and A. Zeller, "Fuzzing with code fragments," Proceedings of the 21st USENIX Security Symposium, pp. 445-458, Aug. 2012.
〔6〕 Google Project Zero, "fuzzilli" https://github.com/googleprojectzero/fuzzilli, Mar. 18, 2020
〔7〕 Hyuk-woo Park, Sung-kook Kim, and Soo-mook Moon, "Work-in-progress: advanced ahead-of-time compilation for javascript engine," Proceeding of the 2017 International Conference on Compilers, Architectures and Synthesis For Embedded Systems, pp. 1-2, Nov. 2017.
〔8〕 Mozilla Fuzzing Security, "funfuzz" https://github.com/MozillaSecurity/funfuzz, Mar. 22, 2020
〔9〕 Google Project Zero, "domato" https://github.com/googleprojectzero/domato, Mar. 23, 2020
〔10〕 P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 206-215, June 2008.
〔11〕 MITRE, "CVE-2019-5782" https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5782, Mar 18, 2020
〔12〕 v8, "v8" https://github.com/v8/v8, Mar. 22, 2020
〔13〕 Min-su Lee, Je-hyun Lee, Ho-bin Kim, and Chan-ho Ryu, "Instrumentation Performance Measurement Technique for Evaluating Efficiency of Binary Analysis Tools," Jonornal of The Korea Institute of information Security & Cryptology, 27(6), pp. 1331-1345, Dec. 2017, 2006.
〔14〕 G. Southern and J. Renau, "Overhead of deoptimization checks in the V8 javascript engine," IEEE International Symposium on Workload Characterization (IISWC), pp. 1-10, Sep. 2016.
〔15〕 N. K. Madhukar, R. Behnam, and H. Ben, "Server-side type profiling for optimizing client-side JavaScript engines," ACM SIGPLAN Notices vol. 51, no.2, pp. 140-153, Oct. 2015.
〔16〕 B. Michael, B. Florian, F. Manuel, L. Francesco, S. Wolfram, T. Nikolai, and V. Herman, "SPUR: a trace-based JIT compiler for CIL." Proceedings of the ACM international conference on Object oriented programming systems languages and applications, pp. 708 - 725, Oct. 2010.
〔17〕 M. Yusuf, A. El-Mahdy and E. Rohou, "On-stack replacement to improve JIT-based obfuscation a preliminary study," Proceedings of the 2nd International Japan-Egypt Conference on Electronics, Communications and Computers, pp. 94-99, Mar. 2014.
〔18〕 J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-Aware Greybox Fuzzing," Proceedings of the 41st IEEE/ACM International Conference on Software Engineering, pp. 724-735, May. 2019.
〔19〕 The Clang Team, "Clang 11 documentation," https://clang.llvm.org/docs/San

itizerCoverage.html, Mar. 22, 2020
〔20〕 Google, “chromium bug 944062” https:/ /bugs.chromium.org/p/chromium/issue s/detail?id=944062, Apr. 29, 2020

〈저자소개〉

김 홍 교 (Hong-Kyo Kim) 학생회원
2018년 2월: 상명대학교 컴퓨터과학과 학사
2018 9월~현재: 고려대학교 정보보호학과 석사과정
〈관심분야〉 정보보호, 시스템 및 네트워크 보안

문 종 섭 (Jong-sub Moon) 종신회원
1981년 2월: 서울대학교 계산통계학과 학사
1983년 2월: 서울대학교 계산통계학과 석사
1991년 2월: Illinois Institute of Technology 전산학과 박사
1993년 3월~현재: 고려대학교 전자 및 정보공학부 교수
2001년 2월~현재: 고려대학교 정보보호대학원 겸임교수
〈관심분야〉 정보보호, 운영체제, 침입탐지