Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.29, NO.2, Apr. 2019 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2019.29.2.275

웹 어셈블리 모듈 안전성 검증을 위한 퍼징 방법

박 성 현,† 강 상 용, 김 연 수, 노 봉 남‡
전남대학교 정보보안협동과정

Fuzzing Method for Web-Assembly Module Safety Validation

Sunghyun Park,† Sangyong Kang, Yeonsu Kim, Bongnam Noh‡
Interdisciplinary Program of Information Security, Chonnam National University

요 약

웹 어셈블리는 웹 브라우저 자바스크립트의 성능 향상을 위해 설계된 새로운 바이너리 표준이다. 웹 어셈블리는 효율적인 실행 및 간결한 표현과 여러 언어를 바탕으로 작성된 코드를 네이티브에 가까운 속도로 구동될 수 있는 새로운 웹 표준으로 자리 잡고 있다. 하지만 현재 웹 어셈블리 취약성 검증은 웹 어셈블리 인터프리터 언어에 제한되어 있으며, 웹 어셈블리 바이너리 자체에 대한 취약성 검증은 부족한 상황이다. 따라서 웹 어셈블리의 자체적인 안전성 검증이 필요한 실정이다. 본 논문에서는 먼저 웹 어셈블리의 구동 방식과 현재 웹 어셈블리의 안전성 검증 방법에 대해서 분석한다. 또한 기존에 발생하였던 웹 어셈블리 안전성 검증 방식에 대해 살펴보고, 이에 따른 기존 안전성 검증 방식의 한계점을 분석한다. 최종적으로 기존 안전성 검증 방법의 한계점을 극복하기 위한 웹 어셈블리 API 기반 퍼징 방법을 소개한다. 이는 기존 안전성 검증 도구로 탐지할 수 없었던 크래시를 탐지함으로써 제안하는 퍼징의 효용성을 검증한다.

ABSTRACT

Web-assemblies are a new binary standard designed to improve the performance of Web browser JavaScript. Web -assemblies are becoming a new web standard that can run at near native speed with efficient execution, concise representation, and code written in multiple languages. However, current Web-assembly vulnerability verification is limited to the Web assembly interpreter language, and vulnerability verification of Web-assembly binary itself is insufficient. Therefore, it is necessary to verify the safety of the web assembly itself. In this paper, we analyze how to operate the web assembly and verify the safety of the current web-assembly. In addition, we examine vulnerability of existing web -assembly and analyze limitations according to existing safety verification method. Finally, we introduce web-assembly API based fuzzing method to overcome limitation of web-assembly safety verification method. This verifies the effectiveness of the proposed Fuzzing by detecting crashes that could not be detected by existing safety verification tools.

Keywords: Web-assembly, Web-assembly Fuzzing, Vulnerability

I. 서 론

웹 어셈블리[1]는 웹 브라우저에서 효율적인 실행과 간결한 표현을 위해 설계된 언어로 안전하고 이식 가능한 저 수준 코드 형식의 표준이다. 웹에서 네이티브 코드에 가까운 성능을 보이기 위해 현대 브라우저에서 구동한 새로운 형식의 코드이며, 이는 웹 어셈블리 이전에도 asm.js 와 같은 기술을 통해 웹에서 구동되는 프로그램의 구동 속도를 향상시키고자 하는 노력이 있었다. 웹 어셈블리는 Google,

Received(11. 08. 2018), Modified(02. 12. 2019), Accepted(02. 15. 2019)
† 주저자, everpall@gmail.com
‡ 교신저자, bbong@jnu.ac.kr(Corresponding author)

Mozila와 같은 주요 웹 브라우저 개발사와 W3C[2]를 중심으로 한 오픈소스 프로젝트이다. 또한 웹 어셈블리는 현재 개발 단계로써 안전성 검증 단계는 웹 어셈블리 인터프리터(자바스크립트)에만 집중되어 있다. 이에 실질적인 웹 어셈블리 모듈 자체에 대한 취약성 검증에 대한 연구는 미비한 수준이다.

웹 어셈블리 언어로 작성된 바이너리는 독립적인 실행이 불가하다. 이는 웹 인터프리터 언어인 자바스크립트의 종속적인 환경으로 인해 웹 어셈블리 모듈이 생성되고 호출되기 때문이다. 이러한 원인은 웹 어셈블리 취약성 검증을 어렵게 만드는 요소 중 하나이다. 결국 웹 어셈블리 바이너리의 자체의 독립적인 실행이 불가하기 때문에 취약성 검증 도구의 활용이 어렵다. 이는 기존의 취약성 검증 도구를 활용 하더라도 자바스크립트를 통해 할당된 웹 어셈블리 모듈 자체에 대한 검증은 쉽지 않음을 나타낸다. 또한 현재 개발 단계의 웹 어셈블리 언어의 특성상 안전성 검증을 위한 초기 시드 파일 확보가 어렵다는 점도 취약성 검증에 대한 어려움의 이유 중 하나가 된다.

본 논문에서는 시드 코퍼스 관리(Seed 코퍼스 Management)를 통한 웹 어셈블리 바이너리 집합 생성 및 이를 통한 웹 어셈블리 API 기반 퍼징 방법을 제안한다. 먼저 다량의 초기 시드 코퍼스 파일을 확보하기 위해 LibFuzzer[3] 기반의 시드 코퍼스 관리와 그에 따른 유효 시드 파일 획득 전략으로 다량의 유요한 웹 어셈블리 바이너리 시드 집합을 생성한다. 이후 웹 어셈블리 모듈 자체적인 퍼징을 위해 바이너리 파싱 및 정의된 섹션 정보 분석을 진행한다. 최종적으로 웹 어셈블리 모듈 퍼징을 위한 API 구성에 필요한 정보를 추출 후 퍼징을 진행한다. 우리는 실험을 통해 기존의 안전성 검증 도구로 생성하기 어려웠던 크래시 파일을 생성해낼 수 냈으며, 이를 통해 실제로 해당 도구가 취약성 탐지에 활용이 가능함을 보였다.

논문의 구성은 다음과 같다. 2장에서는 관련 연구를 통한 웹 어셈블리 개요 및 기존 안전성 검증 도구의 한계점을, 3장에서는 제안하는 분석 메커니즘을, 4장에서는 제안하는 방법에 따른 웹 어셈블리 API 기반 바이너리 퍼징에 대한 실험 및 평가를, 5장에서는 결론을 소개한다.

II. 관련 연구

2.1 웹 어셈블리 개요

2.1.1 웹어셈블리 개요

웹 어셈블리(WebAssembly)는 자바스크립트가 아닌 다른 프로그래밍 언어로 작성된 코드를 브라우저에서 실행시키기 위한 방법으로 효율적인 실행과 간결한 표현을 위해 설계된 안전하고 이식 가능한 저수준 코드형식이다. 주요 목표는 브라우저에서 고성능 응용 프로그램을 활성화 하는 것이다. 현재 웹 어셈블리는 구글, 파이어폭스, 엣지, 사파리를 포함한 주요 브라우저 제작사와 W3C커뮤니티 그룹에서 표준을 정의하고 개발하고 있다. 2015년에 웹 어셈블리 커뮤니티 그룹이 생성되어 개발을 진행 2018년 현재는 최소 기능 제품이 배포되어 구글 크로미움(Google Chromium)[4]과 파이어폭스 브라우저에서 실험적으로 사용이 가능한 상태이다.

2.1.2 웹 어셈블리 컴파일

현재 웹 어셈블리를 가장 잘 지원하고 있는 컴파일 도구 체인은 LLVM[5]이다. 많은 수의 프론트엔드와 백엔드가 LLVM으로 변환이 가능하다. 현재 웹 어셈블리도 변환 기능이 추가되어 사용된다. 또한 웹 어셈블리 텍스트 표현(Text Format)을 직접 코딩하여 웹 어셈블리 모듈을 생성하는 것도 가능하다. Fig 1은 웹 어셈블리 컴파일러 툴 체인을 나타낸다. 웹 어셈블리 모듈(wam)는 최초 C 또는 C++ 언어로 작성되고, clang 프론트 엔드를 통해 LLVM IR 로 변환된다. 이후 LLVM 백엔드 환경에서 IR은 wasm(웹 어셈블리 파일 포맷) 으로 변환 작업이 수행된다. wasm 최종적으로 자바스크립트를 통해 웹상에서 실행환경이 구축될 수 있다.

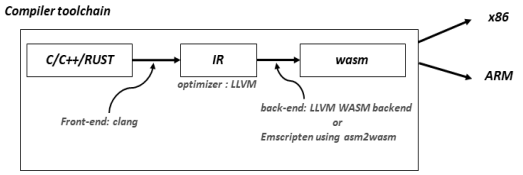


Fig. 1. Webassembly Compiler toolchain

2.1.3 웹 어셈블리 구조

웹 어셈블리는 기본적으로 C파일을 이용하여 생성된다. 다음은 웹 어셈블리로 변환할 C함수이다.

```
// simple_C_func.c
int add42(int num) {
  return num + 42;
}
```

생성된 웹 어셈블리 모듈은 아래와 같은 바이너리 파일이다. 이는 16진수뿐만 아니라 이진 표기법이나 사람이 읽을 수 있는 형식으로 쉽게 변환될 수 있다. 예를 들어 num+42는 Fig 2의 와 같이 표현이 되고, 이는 웹 어셈블리 내에서 실제 코드에 해당되는 내용을 포함한다는 것을 확인할 수 있다.

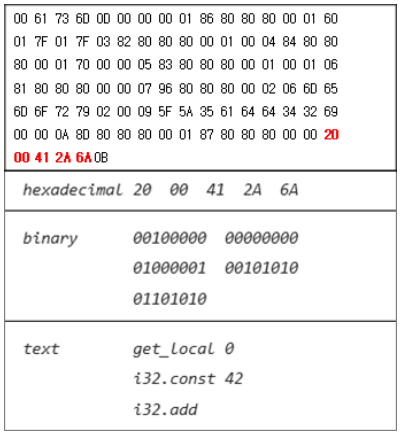


Fig. 2. Webassembly Binary Format

2.1.4 웹 어셈블리 모듈 API

웹 어셈블리는 기본적으로 자바스크립트로 로드되고 컴파일 된다. 자바스크립트는 먼저 웹 어셈블리 모듈 바이트를 Typed Array 또는 Array-Buffer로 가져온다. 다음으로 웹 어셈블리 모듈을 컴파일 한다. 마지막으로, 호출 가능한 함수들을 export 또는 import하고 컴파일 된 웹 어셈블리 모듈을 인스턴스화 한다. table 1은 브라우저상에서 웹 어셈블리를 불러오는 과정을 지원해 주는 자바스크립트 API들을 보여준다. 자바스크립트는 해당 API를 통해 웹 어셈블리 모듈 자체적인 검증을 진행할 수 있고, 모듈 내부의 코드를 사용가능하도록 이식할 수 있다.

Table 1. Webassembly Module API

API	Discription
WebAssembly .Module	Functions that accept Web assembly binaries
WebAssembly .Instance	Functions that instantiate Web assemblies
WebAssembly .compile	Compiling module functions in wemb assembly binary
WebAssembly .Table	Functions to create a table object for the Web assembly table
WebAssembly .Memory	Functions that manage memory in Web assemblies and JavaScript
WebAssembly. RuntimeError	Web assembly runtime error function
WebAssembly .ComplieError	Web assembly compile error function

2.2 웹 어셈블리 안전성 검증 도구

현재 발생하는 대부분의 웹 어셈블리 취약점은 자바스크립트에서 웹 어셈블리 모듈이 아닌 이를 사용하는 과정에서의 발생하는 인터프리터 관련 취약점이었다. 하지만 위와 같은 API를 사용을 통한 자체적인 웹 어셈블리 모듈 안전성 검증은 아직 부족한 상황이다. 웹 어셈블리 모듈 내에서 메모리 오류 또한 빈번하게 발생할 수 있고, 모듈의 인스턴스 및 컴파일 과정에서 취약점이 발생할 수 있기 때문이다. 이에 웹 어셈블리의 모든 실행 과정에서의 안전성 검증을 수행하여 취약점 탐색에 따른 안전성 검증 효율을 높일 방법에 대한 연구가 필요한 실정이다.

2.2.1 LibFuzzer

LibFuzzer는 구글 자바 스크립트 엔진인 v8 내부에서 사용되는 대표적인 오픈소스 기반 퍼저이다. LibFuzzer를 통해 v8[6] 엔진 내의 여러 부분의 소프트웨어 안정성을 검증하는데 사용한다. 웹 어셈블리 또한 LibFuzzer를 이용하여 안전성 검증을 수행할 수 있다.

LibFuzzer는 과정 기반의 퍼징을 수행하고, 코드 커버리지를 높이기 위해 Guided-Fuzzing 방법을 이용하는 진화된 형태의 퍼징 도구이다.

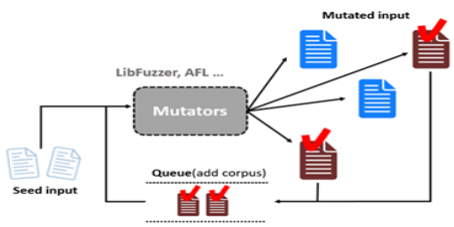


Fig. 3. LibFuzzer fuzzing method

LibFuzzer는 ASAN(Address Sanitizer)[7]과 퍼징을 함께 수행하는 대표적인 도구로써 퍼징을 수행하기 전에 먼저 해당 기술을 적용하여 메모리 사용상의 오류 여부를 점검한다. LibFuzzer는 테스트 중인 라이브러리와 연결되어 있으며, 목표 함수에 도달하는 특정 인풋 값을 라이브러리에 제공한다.

또한 LibFuzzer는 코드의 어느 영역에 도달했는지 추적 범위를 최대화하기 위하여 초기 입력 데이터의 코퍼스(코퍼스)에 변형을 생성하고, 생성된 코퍼스파일은 함께 저장되고 변형 기반(Mutation based) 퍼징을 수행하게 된다.

2.2.2 JsfunFuzz

Jsfunfuzz는 자바스크립트 기반의 퍼징 도구로써 자바스크립트의 문법 구조를 미리 정의하여 자바스크립트 퍼징을 수행 하게 된다. 미리 정의 된 문법 구조를 이용하여 랜덤하게 자바스크립트의 사용자 정의 함수로 구성된 테스트 케이스를 생성한다. 퍼징의 효율을 높이기 위해서 테스트 케이스의 컴파일 및 디컴파일 과정을 이용하며, 웹 브라우저의 취약점을 탐지할 수 있는지 검증한다[8].

자바스크립트에서 문법적인 오류를 발견하기 위한 퍼저로써 지정된 깊이(depth)에 따른 문법 구문을 생성한다. 생성 가능한 문법은 주로 반복문이나 조건문, 스위치 문, 예외 구문 등이다. 생성된 코드에서 문자를 변형하는 등의 간단한 변형을 수행한다. 일반적인 퍼징 도구와는 다르게 문법적인 오류를 발견하기 위한 퍼저이기 때문에 주로 코드를 결합하여 생성하는 방식을 취하며, 수식이나 데이터 대입을 수행하는 코드가 존재한다. Fig.4는 Jsfunfuzz의 기본 알고리즘을 나타낸다.

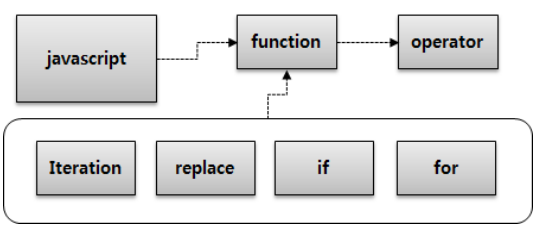


Fig. 4. Jsfunfuzz fuzzing method

2.3 기존 안전성 검증 도구의 한계점 및 개선 방향

2.3.1 LibFuzzer 한계점

LibFuzzer의 경우 시드 파일을 통해 도달하는 경로가 정상적인 경로인지 비정상적인지 판단하고, 새로운 경로라고 생각되면 시드를 추가하게 된다. 하지만 기존의 한정적인 시드 파일을 기준으로 에러를 포함하는 수많은 경로가 코퍼스에 추가되기 때문에 유효한 파일의 생성률은 현저하게 낮아진다. 이는 웹 어셈블리 자체적인 분석에 어려움을 겪는다.

또한 현재 Chromium에서 제공되는 LibFuzzer의 웹 어셈블리 퍼저는 주로 자바스크립트를 통한 웹 어셈블리 모듈을 처리한 이후의 과정에서의 안전성 검증에 초점을 맞추었다.

이에 다양한 샘플을 확보를 위한 방법과 자바스크립트에서 웹 어셈블리 모듈 내의 인스턴스 생성 과정 및 모듈 인스턴스 함수 동작 과정 테스팅에 대한 자체적인 안전성 검증 방안이 필요한 상황이다.

2.3.2 JsfunFuzz 한계점

Jsfunfuzz는 자바 스크립트 퍼징 전용 도구로써 자바스크립트 코드 자체를 생성하여 오류 발생을 유도한다. 기존 퍼저와의 차이점은 자바스크립트가 사용하는 문법 규칙을 퍼저에 주어 코드 생성 시에 문법 오류를 최소화하여 퍼징을 수행한다는 점이다. 하지만 현재 웹 어셈블리는 자바스크립트에서 문법적으로 다양하게 사용되고 있지 않다. 웹 어셈블리는 사용하는 API가 한정적이며, 실제 사용에 필요한 구문이 복잡하지 않기 때문이다.

또한 Jsfunfuzz 자체는 파이어폭스 브라우저에서의 안전성 검증을 수행하기 위해 구현되어있다. 웹 어셈블리와 같은 범용 브라우저에서 실행이 가능한 웹 어셈블리 모듈을 테스트하기에는 부적절하다.

따라서 웹 어셈블리 모듈을 로드하는 경우, 그리고 인스턴스 과정에서의 적절한 안전성 검증이 필요하다. 또한 인스턴스 과정 이후에 웹 어셈블리 API

를 사용하는 모듈 자체적인 안전성 검증이 필요하다.

III. 제안하는 웹 어셈블리 안전성 검증 방법

3.1 샘플 파일 확보를 위한 퍼징

코퍼스 관리 시스템은 다양하고 유효한 웹 어셈블리 시드 집합의 생성을 목적으로 한다. 다양한 형태로 변형된 웹 어셈블리 모듈을 생성하기 위한 변형 기반 퍼저인 LibFuzzer를 기반으로 할 수 있다. 이후 퍼저에 의해 생성된 바이너리에 대한 유효성 검증을 수행하여 실제 퍼징에 사용할 수 있는 웹 어셈블리 파일을 확보하는 것이 목적이다.

3.1.1 초기 시드 집합 확보

웹 어셈블리의 다양한 함수 영역에 도달하기 위해 공개되어 있는 Chromium 소스코드 내에서는 기능 점검을 위한 웹 어셈블리 바이너리 시드를 제공하는데, 이를 시드 파일로 이용하여 초기 시드파일로 사용할 수 있다. 초기 시드파일로 사용할 모듈의 목록은 다음과 같다.

Table 2. Webassembly Initial seed file

address.wasm	index data load & store Test
exports.wasm	export operator Test
f32.wasm	f32 operator Test
f64.wasm	f64 operator Test
float_memory.wasm	floating-point load & store Test
get_local.wasm	get_local operator Test
globals.wasm	global variable Test
call.wasm	call operator Test
i32.wasm	i32 operator Test
i64.wasm	i64 operator Test
func.wasm	func declarations Test
imports.wasm	import operator Test
set_local.wasm	set_local operator Test
stack.wasm	Create stack and access Test

3.1.2 유효 시드 집합 추출 문제

Chromium의 소스 코드 내에는 웹 어셈블리 모듈을 테스트하기 위한 퍼저를 또한 제공하고 있다. 첫 번째로, 해당 퍼저들을 이용해 웹 어셈블리 샘플 파일을 생성하는 방법을 제안한다. 웹 어셈블리 테스팅을 위해 제공하는 퍼저의 종류는 v8_wasm_fuzzer, v8_wasm_code_fuzzer, v8_wasm_compile_ fuzzer등 총 11종의 퍼저가 존재한다. 해당 퍼저들은 초기 시드 집합을 생성하는데 기반이 될 수 있다.

유효 시드 집합 추출은 LibFuzzer를 기반한 wasm 퍼저로 Fig6 과 같은 방식으로 수행된다. 먼저 코퍼스에 파일을 추가를 하여 초기 시드 집합을 생성한다. 이후 커버리지를 높이기 위해 mutation 과정에서 InsertByte, DeleteByte, ChangeByte 등의 방식을 통해 입력 파일을 변형한다. 이와 같은 방식으로 진행 될 경우 변형된 입력 바이너리가 유효하지 않은 웹 어셈블리 모듈로 생성될 확률이 높을 수 있다. 이는 해당 바이너리가 예외처리 경로 및 오류 구문을 모두 포함하기 때문에 유효하지 않은 바이너리가 생성되는 경우가 대부분일 수 있다.

실제 대부분의 wasm_fuzzer에서 Fig 5와 같은 과정을 수행하게 된다. 하지만 코퍼스 파일에는 시드 집합으로 사용될 수 있는 정상적인 wasm 파일의 생성 비율은 매우 낮고, 대부분 유효하지 않는 웹 어셈블리 모듈이 생성되어 시드파일로 재사용 된다. 이러한 문제를 해결하기 위해 유효 시드 집합 생성 문제를 고려하여 유효한 바이너리 생성 비율을 높일 수 있는 전략이 필요하다.

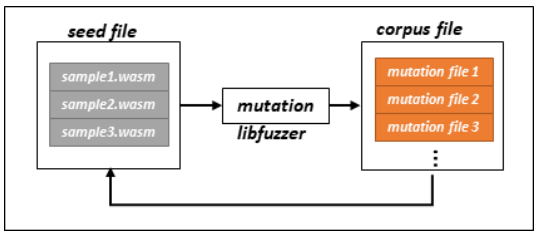


Fig. 5. Generation of corpus through general LibFuzzer

3.1.3 유효 시드 집합 추출 전략

Fig 6은 제안하는 유효 시드 집합 추출 과정을 보여준다. 경로 삭제 방식(path deletion method)을 통해 새로운 경로를 탐색할 시 해당 입력을 새로운 코퍼스로 재생성하는 방식을 응용한다. 이는 초기 시드를 통해 생성된 코퍼스 파일에서 유효한 웹 어셈블리 시드를 관리한다. 유효하지 않은 웹 어셈블리 모듈의 예외처리 구문에서 발생한 코퍼스는 다음 퍼징의 시드로 전달된다. 해당 과정을 통해 예외처리 구문을 따르는 입력의 생성을 최소화 할 수 있다. 즉, 퍼저를 통해 예외처리 경로를 따르지 않는 웹 어셈블리 모듈을 우선적으로 생성함으로써 유효한 웹 어셈블리 모듈의 생성 비율을 높일 수 있다.

Fig 7의 알고리즘은 11종류의 퍼저 실행과 코퍼스의 유효성 검사를 1라운드(line 1~9)로 정의한다. 11종류의 퍼저가 실행이 되고, 실행 과정에서 생성된 전체 코퍼스를 두 단계(line 7,8)에 거쳐 파일의 유효성 검증을 실시한다. 첫 번째로 디버깅 용도로 지원된 텍스트 포맷형식(wat)으로 변환을 한다. 변환을 위해 wabt(The WebAssembly Binary Toolkit)[9]를 활용한다. 텍스트 포맷 형식에서는 웹 어셈블리 모듈이 변형되는 과정(line 7)에서 섹션 데이터가 손상되어 섹션을 구분할 수 없거나, 확인할 수 없는 니모닉 등을 탐지하여 어셈블리 모듈의 유효성 검증을 할 수 있다.

두 번째로 텍스트 포맷으로 변환된 파일을 다시 웹 어셈블리 바이너리 포맷으로 변환하는 과정이다.(line 8) 해당 과정에서는 텍스트 포맷으로는 변환이 되었으나 코드 영역의 오류를 탐지하여 유효한 웹 어셈블리만을 추출한다(line 9). 결과적으로 전체 코퍼스 중 유효한 웹 어셈블리 바이너리는 별도로 추출하여 코퍼스를 구성하는 경로 삭제 전략을 구현한다.

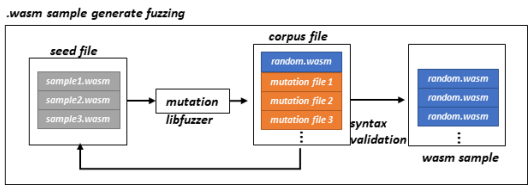


Fig. 6. Proposed 코퍼스 Generation Method

Algorithm Webassembly Module Valid Check & fuzzing

INPUT *C: repeat count*
P_{fuzzer}: Path include fuzzers
p_{seed}: Path include seeds
OUTPUT D_{vaild}: Valid WebAssembly Module Directory

DECLARE *Fuzzer*: wasm_fuzzer
➤ $L_f = \{l_{f(0)}, l_{f(0)}, l_{f(0)}, \ldots, l_{f(0)}\}$
Seed: seed_file(module)
➤ $L_s = \{l_{m(0)}, l_{m(1)}, l_{m(2)}, \ldots, l_{m(n)}\}$
Wasm: valid_wasm_list
➤ $L_{wasm} = \{l_{wasm(0)}, l_{wasm(1)}, l_{wasm(2)}, \ldots, l_{wasm(n)}\}$

BEGIN
1: wasm_fuzzer list of l_0
2: **WHILE** c_n in C **THEN**
3: **WHILE** l_{fuzzer} in L_f **THEN**
4: *Run* l_{fuzzer}
5: $L_s \leftarrow$ newlySeed()
6: **WHILE** l_{module} in L_s **THEN**
7: *wasm2wat*(l_{module}) ➤ valid check wast format
8: *wat2wasm*(l_{module}) ➤ valid check wasm format
9: $L_{wasm} \leftarrow$ get D_{vaild} file list
END

FUNCTION newlySeed
12: L_{old} = get old seed list
13: L_{now} = get now seed list
14: **return** $L_{now} - L_{old}$ ➤ get newly created seed list
END FUNCTION

Fig. 7. Proposed corpus Generation Algorithm

3.2 웹 어셈블리 모듈 테스팅

웹 어셈블리 취약점 분석결과 웹 어셈블리 관련 취약점 중 브라우저 자체에서 웹 어셈블리 로드 및 인스턴스 생성에 대한 안전성 검증의 한계점이 존재한다. 또 인스턴스화 과정뿐만 아니라 웹 어셈블리 관련 API를 불러오는 과정에서의 안전성 검증 또한 기존 퍼징을 이용해서 검증하기 어렵다는 한계점이 존재한다. 이러한 안전성 검증의 한계점을 극복하기 위해 자바스크립트 API 생성 기반 퍼저를 이용하여 검증을 수행한다.

제안 방법인 웹 어셈블리 API 퍼징의 전체적인 개요는 Fig 8와 같다. 먼저 기존에 수집된 유효한

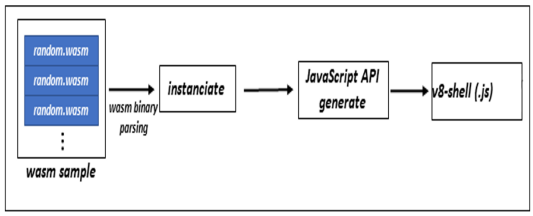


Fig. 8. Overview of Web-assembly API Fuzzing

웹 어셈블리 시드 집합 중에서 각각의 샘플 파일을 파싱하고 필요한 정보를 수집하여 자바스크립트 API를 구성하는데 필요한 정보를 추출한다.

이후 안전성 검증을 위한 인스턴스화(instanciate)를 진행하고, 인스턴스화 이후에 웹 어셈블리 API 함수를 로드하는 과정에서의 안전성 검증을 수행하기 위한 바이너리 파싱 및 해당 정보들을 이용하여 .js 파일을 생성한다. 이 과정에서는 dharma[10] 퍼징 방식을 적용한다. 생성된 .js 파일은 ASAN이 적용된 v8 쉘 실행을 통한 크래시 발생 여부를 확인할 수 있다.

위 과정들은 실제로 웹 어셈블리가 크로미움 v8 shell 에서 동작의 퍼징을 제외한 과정을 일부 유사하게 옮겨놓은 것이다. 만약 v8 shell 에서 크래시가 발생되었을 경우 ASAN 로그를 출력하고, 크래시가 발생되지 않을 경우 이전의 인스턴스화 과정 부터 다시 수행하게 된다.

3.2.1 웹 어셈블리 바이너리 테스트를 위한 모듈 파싱

현재 웹 어셈블리는 자바스크립트로 로드되고 컴파일 된다. 기본적인 로드의 경우 다음과 같은 세 단계가 있다. 첫째, 먼저 웹 어셈블리 모듈 바이트를 Typed Array 또는 Array Buffer로 형태로 가져온다. 둘째, 웹 어셈블리 모듈을 컴파일 한다. 셋째, 호출 가능한 함수들을 export 또는 import 하고 컴파일 된 웹 어셈블리 모듈을 인스턴스화 한다.

첫 번째 단계에서 Typed Array 또는 Array Buffer를 가져오는 여러 가지 방법이 있다. 네트워크를 통해 XHR 또는 fetch API를 사용하거나, indexedDB에서 FILE을 읽어오거나, 자바스크립트로 작성할 수도 있다. 본 연구에서는 자바스크립트로 해동 모듈을 읽어온 후 작성하는 방식을 취하는 방향으로 전개한다. 다음 단계는 WebAssembly.compile 함수를 사용해 바이트를 컴파일 하는 것이다. 이는 보통 WebAssembly.Module 함수를 통해서 반환되는 값을 전달하기도 한다. 마지막으로 컴파일된 모듈에서 전달되는 값은 WebAssembly.Instance 함수를 통해 모듈을 인스턴스화를 진행한다. 해당 함수를 호출하는 과정에서 import 정보를 명시해 주어야 할 때도 있으며, 이후 최종적으로 웹 어셈블리 모듈 내부의 export 함수를 사용할 수 있는 환경을 구성한다.

전체적인 모듈 생성 과정은 위와 같으며, 퍼징을 위해서 웹 어셈블리 모듈을 파싱할 필요가 있다. 웹 어셈블리는 총 12종류의 섹션을 갖고, 각각의 섹션은 모듈 내부의 함수의 변환 값으로 사용될 수 있다. 바이너리 파싱 과정을 통해 다양하게 분류되고 정의된 섹션을 분석하고, 자바스크립트 API 구성에 필요한 정보만을 추출하여 사전 파일 형태로 파싱된 정보

Algorithm 2 Webassembly Module Parsing & Create Grammars

INPUT R_{dump}: Webassembly module dump program
W: Target Webassembly module
OUTPUT F_{dg}: Webassembly Module Dictionary file
DECLARE W_{sig}: Webassembly signature
W_{ver}: Webassembly Version

BEGIN
1: $F_{info} \leftarrow R_{dump}(W)$ ➤ read Webassembly sections info
2: $F_{detail} \leftarrow R_{dump}(W)$ ➤ read Webassembly sections detail
3: **WHILE** F_{info} not equal eof **THEN**
4: $L_{info} \leftarrow$ Read F_{info} by section ➤ S_{info} = { name, size, count }
5: $F_{dg} \leftarrow$ createfile(F_{dg}) ➤ create Webassembly dictionary file
6: **WHILE** F_{detail} not equal eof **THEN**
7: $S_{name} \leftarrow$ Read F_{detail} by section name
8: **IF** L_{info}[name] equal S_{name} **THEN** ➤ $L_{section}$ = {type, import, ... }
9: parsing#S_{name}#section(F_{detail}, L_{info})
END

FUNCTION parsing#sectionname#section(F, L)
11: $S_{data} \leftarrow$ Read F by signature data
12: **FOR** item in L[count]
13: $S_{item} \leftarrow$ Read S_{data} by section item
14: writedictionary(F_{dg}) ➤ write Webassembly dictionary file
END FUNCTION

Fig. 9. Web assembly binary parsing algorithm

를 가공한다.

추출한 정보에는 모듈의 익스포트, 임포트 함수 목록과 해당 함수들을 사용하기 위한 함수 파라미터 타입, 테이블과 메모리의 초기 할당 크기 등 웹어셈블리와 관련된 자바스크립트 API를 사용하기 위해 필요한 정보들이 저장된다.

Fig 9의 알고리즘은 API 퍼징을 위한 사전 단계로 웹 어셈블리 모듈 파싱 및 퍼징을 위한 사전 파일을 제작한다. 각각의 웹 어셈블리의 섹션 정보와 섹션의 세부 정보를 획득한 후(line 1,2) 각 섹션의 name, size, count 정보를 저장한다(line 3, 4). 이후 해당 섹션에 해당하는 세부정보 또한 추출한다(line6~9). 파싱과정에서 획득한 섹션 내부 데이터는 웹 어셈블리 사전 파일로 저장되고 각 섹션별 사전 파일은 다음과 같은 형태로 관리될 수 있다. Fig 10의 사전 파일 정보는 import 섹션 정보의 예시이며 해당 import 함수의 파라미터 및 리턴타입 정보를 획득할 수 있다.

```
type_param :=
        +common:int+, +common:int+
        +common:int+
        +common:int+, +common:int+, +common:int+
type_ret :=
        +common:int+
imp_namespace :=
        env
imp_func :=
        abort
        fflush
        fprintf
imp_global :=
        stderr
tbl_init :=
        0
mem_init :=
        1
exp_func :=
        __assert_fail
exp_mem :=
        memory
```

Fig. 10. Webassembly Section dictionary file

3.2.2 웹 어셈블리 API 안전성 검증

웹 어셈블리 API 안전성 검증은 웹 어셈블리 바이너리 파싱을 통한 기본적인 구조 분석을 이용하여 데이터를 가공한 후 자바스크립트 문법에 맞는 statement를 생성함으로써 검증할 수 있다. 본 논문에서는 자바스크립트 API 퍼징 statement를 구성하기 위해서 dharma 퍼징 프레임[10] 워크를 사용한다.

dharma에서 웹 어셈블리 모듈을 테스팅 하기 위해서는 이전 단계에서 파싱한 정보를 바탕으로 웹 어셈블리의 각 섹션에서 생성 가능한 API 문법을 정의해주어야 한다.

문법 체계는 WebAssembly documentation에 정의된 JS API 문법을 따르며, 생성된 statement의 유효율을 높이기 위한 시퀀스 또한 정의한다. 각 섹션별로 별개의 파일(.dg)로 문법을 정의하며, 이는 파일(wasm_err.dg, wasm_export.dg, wasm_import.dg, wasm _mem.dg, wasm_table.dg, wasm_valid.dg)로 구성하였다. 해당 문법을 정의함으로써 웹 어셈블리 API에 사용되는 export, import, memory, table 섹션에 관한 안전성 검증 수행이 가능하게 된다.

IV. 실험 및 평가

본 장에서는 제안하는 API 퍼징 프레임워크를 이용한 웹 어셈블리 안전성 검증 방법을 테스팅 하기 위한 실험 및 평가를 수행한다. 먼저 기존 LibFuzzer의 방법보다 제안하는 시드 파일 확보 방법이 효율성이 있는지에 대해서 실험을 수행한다. 또 웹 어셈블리 API 퍼징을 통해서 실제 취약점을 찾을 수 있는지에 대해 검증한다. 두 번째의 경우 기존 취약점의 재연을 통해 검증을 수행하여 도구의 효용성에 대해서 검증한다.

4.1 유효한 웹 어셈블리 시드 파일 생성

실제 Libfuzzer(wasm-fuzzer)를 통해 유효한 웹 어셈블리 유효 시드 파일 생성의 효율을 검증하고 퍼징에 사용될 초기 시드 파일을 확보할 수 있다.테스트 환경은 Ubuntu 16.04 (64bit), Chromium 61.0.3142.0 환경이다.

LibFuzzer를 통해 생성된 전체 코퍼스 중 유효한 웹 어셈블리 파일의 비율과 제안하는 경로 삭제 방식을 통해 생성된 유효한 웹 어셈블리 모듈의 비율을 비교하여 제안하는 방식의 효율성을 검증한다.

각 실험은 다양한 시드 파일의 생성을 목적으로

하여 약 20시간의 실행을 통해 테스트가 진행하였으며, 각 방식을 통해 생성되는 전체 파일 중 유효한 웹 어셈블리 모듈이 차지하는 비율을 비교하였다.

실험 결과는 table 3의 내용과 같다. 기존 wasm fuzzer는 Chromium에서 제공하는 wasm_ fuzzer이다. Normal Fuzzer를 사용한 경우 102,302개의 코퍼스가 생성되었으며, 이 중 4.9%인 5,100개의 유효한 웹 어셈블리 모듈이 생성되었다. 제안하는 방식의 경우, 162,673개의 코퍼스가 생성되었으며, 이 중 23%인 37,511개의 유효한 웹 어셈블리 모듈이 생성되었다. 실험 결과, 제안하는 경로 삭제 전략을 통해 생성되는 전체 코퍼스의 수가 60% 증가하였으며 유효한 웹 어셈블리 모듈의 개수 역시 기존의 방법에 비해 4.6배의 높은 효율을 보였다.

Table 3. Web Assembly Generation Efficiency

	Nomal wasm fuzzer		Proposed wasm fuzzer	
	count	rate (%)	count	rate (%)
Corpus	102,302	100	162,673	100
valid wast	6,333	6.1	45,638	28.0
Valid wasm	5,100	4.9	37,511	23.0

4.2 웹 어셈블리 안전성 검증을 위한 API 기반 퍼징

웹 어셈블리 API 안전성 검증을 위해서 기존 취약점을 재연하는 방식으로 테스트를 진행하였다. 다음은 제안하는 웹 어셈블리 API 테스팅을 위한 dharma를 통한 퍼즈된 자바스크립트 파일 중 하나이다.

생성된 자바스크립트 파일(.js)을 ASAN이 적용된 v8_shell(d8)에 전달함으로써 v8 내부에서 발생하는 웹 어셈블리 API 기반 크래시 탐색이 가능하다.

table 4 는 기존 도구와 제안하는 도구의 같은 시간 내에 크래시 발생 여부의 결과이다. 본 연구에서 제안한 도구에서 제된 시간에 더 많은 자바스크립트 Crash를 생성함으로써 제안하는 웹 어셈블리 API 퍼저의 효용성을 입증할 수 있었다. 하지만 제안하는 퍼저 또한 결국 기존의 무작위 변형의 한계점

```
 1 var bytes = new Uint8Array('00 61 73 6d 01 00 00 0
0 01 06 01 60 01 7f 01 7f 03 02 01 00 07 09 01 05 5
 f 6d 61 69 6e 00 00 0a 02 01 40 10'.split(/[WsWrWn]+
/g).map(v => parseInt(v, 16)));
 2
 3 try { var myModule = new WebAssembly.Module(byte
s); } catch(e) {}
 4 try { var myInstance = WebAssembly.Instance(myModu
le); } catch(e) {}
...

 25 try { table_obj1 = new WebAssembly.Table( {initia
l:3, maximum:121, element:"anyfunc"} ); var exp_tabl
  e = myInstance.exports.NULL; exp_table.get(8)(79000
4954) } catch (e) { }
```

Fig. 11. Result of Fuzzed Script File(.js)

Table 4. Web Assembly Generation Efficiency

Fuzzer	Times	Crash
Nomal wasm-fuzzer	20 hour	2
Proposed wasm-fuzzer	20 hour	45

을 갖고 있기 때문에 항상 동일한 실험 결과가 나오지는 않을 수도 있다.

V. 결 론

웹 어셈블리는 현대 웹브라우저에서 동작하는 새로운 형식의 코드이며, 여러 언어로 작성된 코드를 네이티브에 가까운 속도로 웹에서 구동이 가능하다는 장점을 가지고 있다. 즉, 웹 어셈블리로 인해 개발자가 원하는 언어로 개발한 애플리케이션을 웹 브라우저 내에서 동작시킬 수 있게 되었다. 이는 지난 10년간 웹 플랫폼의 가장 큰 발전이라고 평가받고 있으며, 여러 브라우저 개발자들에 의해 현재까지도 표준화가 계속해서 진행 중이다.

본 연구에서는 이렇게 차기 RIA 모듈로 주목받고 있는 웹 어셈블리에 대한 안전성 검증 연구를 수행하였다. 먼저 웹 어셈블리의 장단점 및 동작 매커니즘을 분석하고, 파일 포맷에 대한 정리를 수행하였다. 또한 웹 어셈블리 안전성 검증을 위한 기존의 퍼저에 대한 분석을 수행하였다.

분석 내용을 바탕으로 웹 어셈블리 모듈 안전성

검증을 위한 웹 어셈블리 API 기반 퍼징 방식을 제안하였다. 먼저 다양한 웹 어셈블리 바이너리 생성을 위한 퍼징을 수행한다. 이 과정에서 유효한 웹 어셈블리 바이너리 생성 비율을 높이기 위한 전략으로 경로 삭제 방식 알고리즘(path deletion method)를 제안하였고 이는 기존 방법보다 4.6배 높은 효율성을 증명하였다.

이후 웹 어셈블리 생성 기반 API 퍼저(generation-based API fuzzer)를 작성하여 웹 어셈블리와 관련된 자바스크립트 API를 생성하는 퍼징을 진행하였다. 이를 위해 웹 어셈블리 바이너리를 파싱하여 API에 필요한 정보를 획득하였고, 생성된 웹 어셈블리 API의 유효 비율을 높였다. 또한 발생한 크래시 중 기존 퍼저보다 더 많은 비율의 Crash를 탐지함으로써 퍼저의 취약점 탐지 도구로의 실제 활용 가능성을 증명하였다.

References

〔1〕 MDN web docs, https://developer. mozilla.org/ko/docs/WebAssembly/Concepts, 2019

〔2〕 World Wide Web Consortium, https://www.w3.org/, 2019

〔3〕 libFuzzer,https://llvm.org/docs/LibFuzzer. html, 2019

〔4〕 Chromium Projects, https://www.chromium.org/, 2019

〔5〕 Lattner, Chris, and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization. IEEE Computer Society, pp. 75, 2004.

〔6〕 https://github.com/MozillaSecurity/funfuzz/tree/master/src/funfuzz/js/jsfunfuzz, 2019

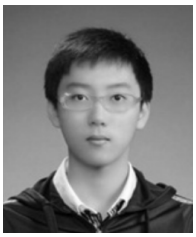
〔7〕 Serebryany, Konstantin, et al. "AddressSanitizer: A Fast Address Sanity Checker." Presented as part of the 2012 {USENIX} Annual Technical Conference, pp. 309-318, Dec, 2012

〔8〕 jsfunfuzz, https://github.com/MozillaSecurity/funfuzz/tree/master/src/funfuzz/js/jsfunfuzz, 2019

〔9〕 wabt, https://github.com/WebAssembly/wabt, 2019

〔10〕 dharma, https://github.com/MozillaSecurity/dharma, 2018

〈저자소개〉



박 성 현 (Sunghyun Park) 학생회원
2016년 8월: 전남대학교 컴퓨터정보통신공학 공학사
2018년 2월: 전남대학교 정보보안협동과정 이학석사
2018년 3월~현재: 전남대학교 정보보안협동과정 박사과정
〈관심분야〉 취약점 분석, 악성코드 탐지, 시스템 보안

강 상 용 (Sangyong Kang) 학생회원
2014년 8월: 전남대학교 컴퓨터정보통신공학 공학사
2016년 8월: 전남대학교 정보보안협동과정 이학석사
2016년 9월~현재: 전남대학교 정보보안협동과정 박사과정
〈관심분야〉 소프트웨어 취약점 분석 및 탐지, 시스템 보안

김 연 수 (Yeonsu Kim) 학생회원
2017년 8월: 전남대학교 소프트웨어공학 공학사
2017년 9월~현재: 전남대학교 정보보안협동과정 석·박사통합과정
〈관심분야〉 사이버 위협 인텔리전스, 머신러닝, 침입탐지

노 봉 남 (Bongnam Noh) 종신회원
1978년: 전남대학교 수학교육과 학사
1982년: KAIST 대학원 전산학과 석사
1994년: 전북대학교 대학원 전산과 박사
1983년~현재: 전남대학교 전자컴퓨터공학부 교수
2000년~현재: 전남대학교 시스템보안연구센터 소장
〈관심분야〉 정보보안, 시스템 및 네트워크 보안