웹 어셈블리의

Just-In-Time Compiled Code 보호 기술 개발

부산대학교 정보컴퓨터공학부

지도교수: 권동현

팀 명 : 인디고 블랙 파스타

팀 원 : 201924486 배명진

202155569 신채원

202155603 정윤서

목차

1. 연구 개요

1.1 과제 배경

1.1.1 WASM Engine

웹 어셈블리(WebAssembly, WASM)는 웹에서 고성능 애플리케이션을 실행하기 위한 안전하고 이식 가능한 이진 명령어 형식을 말한다. WASM 은 브라우저에서 네이티브 코드 실행 속도와 가까운 실행 속도를 제공하며, C, C++, Rust 등 다양한 프로그래밍 언어로 작성된 코드를 웹에서 직접 실행할 수 있게 한다. 이는 웹 개발자들이 복잡하고 계산이 많이 소모되는 작업을 웹 애플리케이션에서 할 수 있도록 한다.

WASM 의 주요 특징 중 하나는 플랫폼에 독립적이라는 점이다. 즉, 동일한 WASM 형식이 다양한 운영체제와 브라우저에서 동일하게 실행될 수 있다. 이 때, WASM 은 하드웨어에 독립적이므로 최신 아키텍처(데스크탑, 모바일, 임베디드 시스템)에서 실행 가능하며, 프로그래밍 언어이나 플랫폼에서도 독립적이기 때문에 특정 프로그래밍 언어나 실행 환경에 우선 권한을 부여하지 않는다. 이러한 특성은 웹 개발의 표준을 확장하고 웹 애플리케이션의 성능을 크게 향상시킨다.

또한, WASM 은 효율적이고 빠르다는 특징이 있다. WASM 은 일반적인 텍스트나 네이티브 코드 형태보다 훨씬 압축된 형태의 이진 코드를 생성하므로, 네이티브 코드 성능에 가까운 성능으로 실행된다. 나아가 디코딩 및 컴파일을 병렬처리 할 수 있으며, JIT(Just-In-Time) 또는 AOT(Ahead-of-Time) 컴파일을 사용하면 네이티브 코드나 바이트 코드를 빠르게 컴파일 가능하다는 장점도 있다.

WASM 을 실행하기 위해서는 WASM 엔진이 필요하다. WASM 엔진은 WASM 모듈을 로드하고 실행하며, 이 과정에서 이진 코드를 해석하거나 컴파일하여 네이티브 코드로 변환한다. WASM 엔진은 브라우저에 내장되어 있으며, 대표적인 엔진으로는 구글의 'V8', 모질라의 'SpiderMonkey', 마이크로소프트의 'Chakra', 그리고 웹킷의 'JavaScriptCore' 등이 있다. WASM 엔진은 최적화 기법을 사용하여 WASM 코드를 효율적으로 실행하며, JIT(Just-In-Time) 컴파일 기술을 활용하여 성능을 극대화한다.

1.1.2 JIT Compile

JIT(Just-In-Time) 컴파일은 프로그램을 실제 실행하는 도중에 기계어 코드로 변환하는 기술이다. 인터프리터 방식은 실행 중에 프로그래밍 언어를 읽어가며 해당

기능에 대응하는 기계어 코드를 실행한다. 이에 반해 정적 컴파일은 실행하기 전에 프로그램 코드를 기계어로 변환한다. JIT 컴파일은 이 두 방식을 혼합한 방식으로 생각할 수 있다. 프로그램 실행 중에 필요한 부분만 실시간으로 컴파일하는 방식으로, 코드 최적화와 실행 속도 면에서 큰 이점을 제공한다.

인터프리터 방식은 최적화 과정 없이 번역하므로 성능이 낮을 수 있으며, 정적 컴파일 방식은 실행 전 컴파일을 해야 하므로 환경에 따라 시간이 오래 걸릴 수 있다. 이에 반해 JIT 컴파일을 사용하면 코드를 실시간으로 최적화할 수 있다. JIT 컴파일러는 바이트 코드를 읽어 빠른 속도로 기계어 코드를 생성할 수 있다. 이 과정은 실시간으로 일어나며, 전체 코드 중 필요한 부분만 변환한다. 또한, 기계어로 변환된 코드는 캐시에 저장되기 때문에 재사용시 다시 컴파일을 할 필요가 없다.

또한, 플랫폼의 독립성을 가진다. JIT 컴파일러는 다양한 플랫폼에서 동일한 바이트 코드를 실행할 수 있게 한다. 이는 프로그램을 여러 플랫폼에 배포할 때, 플랫폼별로 별도의 컴파일 과정을 거치지 않아도 된다는 것을 의미한다.

1.1.3 Google v8

Google V8 은 구글이 C++로 개발한 JavaScript 및 WASM 의 오픈소스 엔진이다. 크롬 브라우저와 Node.js 런타임 환경에서 사용된다. V8 엔진은 JavaScript 와 WASM 코드를 빠르고 효율적으로 실행하기 위해 설계되었으며, 최신 JIT 컴파일 기술을 활용하여 뛰어난 성능을 제공한다. 또한 x64, IA-32, ARM 프로세서를 사용하는 Windows, macOS, Linux 시스템에서 동작한다.

V8 엔진은 코드 실행 시 JIT 컴파일을 사용하여 JavaScript 및 WASM 코드를 네이티브 기계어 코드로 변환한다. 이를 통해 실행 속도를 극대화하고, 다양한 최적화 기법을 적용하여 성능을 향상시킬 수 있다. JIT 컴파일의 과정은 다음과 같다.

먼저 JavaScript 또는 WASM 코드는 먼저 바이트 코드로 변환된다. 이는 V8 엔진이 코드를 효율적으로 분석하고 처리할 수 있도록 합니다. 이후에 V8 엔진은 런타임 과정 중에 프로파일링을 진행한다. 프로파일링은 함수나 변수의 호출 빈도와 같은 데이터를 모으는 과정인데, 이를 통해 자주 실행되는 코드의 지점을 찾을 수 있으며, 최적화 대상 코드를 찾을 수 있다. 이후 프로파일링 결과를 바탕으로, 자주 실행되는 코드 지점을 기계어 코드로 JIT 컴파일을 수행한다. 이 과정에서 최적화 기법이 적용되어 코드의 실행 속도가 향상된다.

이러한 JIT 컴파일 과정을 통해 V8 엔진은 JavaScript 및 WASM 코드를 빠르고 효율적으로 실행할 수 있으며, 이를 통해 웹 애플리케이션의 성능을 극대화할 수 있다.

1.2 과제 목표

1.2.1 MPK key 의 다양화

- MPK Key 할당의 효율성 증대

  16 개의 MPK 를 최대한 활용하여 다양한 메모리 영역에 대한 접근 권한을 효율적으로 설정한다. 메모리 사용 패턴을 분석하여 주요 메모리 영역을 식별하고, 각 메모리 영역의 보안 요구사항에 따라 적절한 키를 사전에 할당한다.

- 동적 키 할당

  런타임 동안 동적으로 MPK 를 할당하고 해제하는 매커니즘을 도입하여 키의 활용성을 극대화한다. 사용되지 않는 키를 자동으로 회수하고, 새로운 메모리 영역에 재할당할 수 있는 전략을 수립한다.

1.2.2 안전한 key 분배를 위한 key 결정 알고리즘 개발

- 보안 기반 키 분배 알고리즘 설계

  메모리 영역의 보안 요구사항(예: 읽기 전용, 쓰기 가능, 실행 가능)에 따라 키를 분배하는 알고리즘을 설계한다.

- 충돌 방지 및 일관성 유지

  메모리 영역 간의 키 충돌을 방지하기 위한 충돌 회피 알고리즘을 개발한다. 다중 스레드 환경에서도 일관된 키 분배를 보장하기 위한 동기화 매커니즘을 도입한다.

2. 요구 조건 분석과 제약 사항

2.1 요구 조건 분석

2.1.1 소스 코드 분석

- v8 엔진의 기능 중 JIT Compile 관련 코드 분석
- 주요 함수 및 코드의 동작을 아키텍처 별로 분석

2.1.2 Intel MPK 다양화

- 생성된 machine 코드 권한 관리 방법 다양화

2.1.3 새로운 보안 feature 고안

- 기존 코드에서 새로이 추가 가능한 보안 feature 고안

2.2 현실적 제약 사항 및 대책

2.2.1 제약 사항

- 키 리소스 제한

  Intel MPK 가 제공하는 16 개의 키는 제한된 자원이다. 다양한 메모리 영역에 대해 다양한 권한을 설정하려면 이 제한된 키를 효율적으로 사용해야 한다.
- 보안 취약점 노출 가능성

  MPK 를 사용하여 메모리 접근을 제어할 때, 잘못된 설정이나 버그로 인해 의도치 않게 민감한 데이터에 대한 접근 권한이 부여되는 등의 보안 취약점이 발생할 수 있다.

2.2.2. 대책

- 키 재사용 및 관리

사용 중인 메모리 보호 키를 효율적으로 관리하고 재사용할 수 있도록 한다. 키 할당 및 해제를 동적으로 관리하는 매커니즘을 도입하여 필요할 때만 키를 할당하고, 사용이 끝나면 즉시 해제한다.

- 보안 취약점 관리

  정기적인 코드 리뷰와 보안 점검을 통해 MPK 설정 관련 취약점을 식별하고 해결한다. 메모리 접근 권한 설정을 엄격하게 관리하고, 의도치 않은 접근을 방지하기 위한 추가적인 검증 절차를 도입한다.

3. 연구 계획

3.1. MPK key 다양화

3.1.1 코드스페이스 관리 구조체 분석

v8 소스코드에서 코드스페이스에 대한 정보를 관리하는 구조체를 찾아 그 구조와 동작을 분석한다. 특히 코드스페이스의 권한을 수정하는 부분의 동작과, 사용하는 변수명 등을 분석하여 동작을 수정할 수 있도록 한다.

3.1.2 key 관리 구조체 구조 분석 및 수정

코드스페이스의 key 를 관리하는 구조체를 찾아 그 구조와 동작을 분석한다. 1 로 하드코딩되어 있는 key 변수를 수정하여 코드스페이스가 여러 개의 key 를 가질 수 있도록 한다. 또한 Intel 아키텍처의 이해를 통해 본 프로젝트에서 사용할 수 있는 key 번호를 알아 아키텍처의 동작에 문제가 없도록 한다.

3.2. key 결정 알고리즘 구현

Intel MPK 의 key 는 16 개밖에 없으므로, 코드스페이스의 개수가 늘어날수록 같은 key 를 가지는 코드스페이스가 생길 수 있다. 이 경우, 같은 key 를 가진 코드스페이스가 컴파일 중이고 쓰기 가능한 권한을 가지면 코드스페이스의 코드가 이미 컴파일 완료되어 써진 상태임에도 불구하고 쓰기 가능한 권한을 가지게 된다. 이러한 상황은 보안 취약점으로 이어질 수 있다.

따라서 공격자가 코드스페이스의 key 를 예상하지 못하도록 하는 key 결정 알고리즘을 고안한다. 고안한 key 결정 알고리즘의 오버헤드를 측정하고, 그 중 성능과 보안성이 적당히 균형적인 것을 채택하여 구현한다.

3.3. 추가적인 보안 기능 고안

위의 계획에 더하여, 코드스페이스 위치 랜덤화 등 MPK 를 이용한 것 외에도 JIT Compile 과정에서 더욱 효율적이고 안전하게 코드스페이스를 보호할 수 있는 기능을 연구한다.

4. 추진 일정과 담당 업무

4.1 연구 일정

주요일정	6 월	7 월	8 월	9 월	10 월
소스코드 분석					
취약점 분석					
중간보고서 및 중간평가표 제출					
연구 마무리 및 최종 평가					
프로젝트 디버깅					
최종보고서 및 최종평가표 제출					
졸업과제 발표 심사					
결과물 업로드					

4.2 담당 업무

배명진

웹 어셈블리 처리 엔진 v8 에서 JIT compile 관련 github 소스코드를 분석한다. v8 에서 WASM 코드를 JIT 컴파일하여 Machine 코드를 생성하는 부분의 소스코드를 분석한다. 이 과정에서 생성된 Machine 코드는 writable 한 권한을 가지게 되는데, 이로 인해 발생할 수

있는 문제점에 대해 탐구한다. 또한, 생성된 Machine 코드의 권한을 Intel MPK(Memory Protection Key)를 활용하여 각각의 page 에서 관리할 수 있는 방식에 대해서 연구한다.

신채원

chrome 의 웹 어셈블리 및 js 엔진인 v8 의 기능 중, JIT Compile 에 관련된 코드를 분석한다. v8 코드에서 JIT Compile 을 하는 주요 함수 및 코드의 동작을 아키텍처 별로 알고 해당 동작에서 발생할 수 있는 보안 취약점을 분석한다. 또한 기존 코드에서 새로이 추가할 수 있는 보안 feature 을 고안한다.

정윤서

v8 엔진 내에 구현된 Just-In-Time compilation 관련 소스 코드를 분석하면서 JIT compilation 을 통해 생성된 웹 어셈블리 머신 코드의 취약점을 분석한다. 이후에 Intel MPK 를 활용하여 key 의 다양화 기능을 구현한다.

5. 참고문헌

https://webassembly.github.io/spec/core/intro/introduction.html

https://ko.wikipedia.org/wiki/JIT_%EC%BB%B4%ED%8C%8C%EC%9D%BC

https://ko.wikipedia.org/wiki/V8_(%EC%9E%90%EB%B0%94%EC%8A%A4%ED%81%AC%EB%A6%BD%ED%8A%B8_%EC%97%94%EC%A7%84)

https://github.com/v8/v8