Pusan National University
Computer Science and Engineering
Technical Report 2024-10

웹 어셈블리의

Just-In-Time Compiled Code 보호 기술 개발

조 이름	인디고블랙파스타	
조원	학번	이름
	202155569	신채원(조장)
	201924486	배명진
	202155603	정윤서
분과명	하드웨어, 보안 (D)	
번호	52	
지도교수	권동현 교수님	

목 차

1. 서론

1.1. 연구 배경

웹어셈블리 (이하 Wasm)는 최신 웹 브라우저에서 실행할 수 있는 새로운 유형의 어셈블리 언어이다.[1] Wasm은 C, C++, Rust 다양한 언어를 컴파일하여 생성될 수 있으며, 다양한 런타임과 브라우저에서 실행된다. 다양한 언어로 작성한 코드를 번거로운 절차 없이도 한 번에 실행할 수 있고, 웹에 특화된 프로그래밍 언어 없이도 웹용 프로그램을 작성할 수 있는 편리성 때문에 Wasm을 사용하는 애플리케이션이 점차 많아지고 있다. 이러한 흐름에 따라 chrome, safari, edge 등 웹 브라우저에서도 점차 Wasm을 지원한다.

사용자와의 실시간 소통을 필요로 하는 웹용으로 개발되었기 때문에, Wasm에 있어서 성능은 중요한 요소이다. Wasm 코드에 대한 효율적인 실행을 위해서, 웹브라우저의 Wasm 런타임은 최근에 Just-in-Time (JIT) 컴파일 기능을 지원하고 있다. JIT 컴파일러는 Wasm 코드에 대한 실행 요청이 처음 발생했을 때 Wasm 코드를 수행 중인 기기에 해당하는 기계어로 변환하고, 이 기계어를 수행한다. 이후 동일한 Wasm 코드에 대한 실행 요청이 들어오면 추가적인 컴파일 과정 없이 생성해둔 기계어 코드를 바로 수행하는 방식으로 동작하여 기존의 Interpreter 대비 빠른 수행을 보장한다.

1.2. 기존 문제점

Wasm 코드에 대한 JIT 컴파일 방식은 새로운 보안 위협을 야기한다. JIT 컴파일된 코드는 웹 어플리케이션 동작 중에 생성되고 수행되어야 하기 때문에 이를 지원하기 위해서 JIT 컴파일된 코드가 저장된 메모리 공간은 쓰기 가능하고 동시에 수행 가능해야 한다. 이는 오늘날 널리 쓰이는 메모리 보호 기법인 W^X 정책을 위반한다[2]. 즉 공격자가 웹 브라우저 내에 있는 메모리 취약점들을 익스플로잇하여 해당 JIT 컴파일된 코드를 변조할 경우 웹 브라우저를 대상으로 코드 삽입/변조 공격 등을 수행할 수 있게 된다.

이에 기존 웹 브라우저들은 JIT 컴파일된 코드가 저장된 메모리 페이지에 대한 접근 권한을 동적으로 관리함으로써 이러한 위협에 대응하였다. 그들은 JIT 컴파일된 코드가 저장된 메모리 페이지에 대한 쓰기 권한을 빼앗고, JIT 컴파일 과정 혹은 런타임 수행 종료 후 코드 삭제 시와 같이 해당 메모리 페이지에 대한 쓰기 연산을 실행해야 할 때만 쓰기 권한을 일시적으로 부여해 JIT 컴파일된 코드에 대해 W^X 보호를 제공하였다. 예를 들어 Wasm 코드에 대한 JIT 컴파일 기능을 제공하는 웹브라우저 중 google의 v8은 Intel 사의 MPK를 이용하여 JIT 컴파일된 코드에 대한 접근 권한을 제한하였고[3], Safari의 JavaScriptCore는 페이지 테이블에 기반한 방법으로 접근 권한을 제한하였다[4]. 그러

나 V8 런타임의 보호 정책에서는 모든 코드스페이스의 접근 권한을 하나의 보호 키로 관리하여, 하나의 코드스페이스에 대한 JIT 컴파일이 수행될 때 모든 코드스페이스가 쓰기 권한을 가진다는 문제점이 있다. 이는 공격 표면을 넓혀 보안상의 문제를 야기한다. 또한 JavaScriptCore에서 채택한 페이지 테이블 기반 보호 기술은 하드웨어 기반의 메모리 보호 기술보다 성능 오버헤드가 현저히 높아 웹브라우저에서 요구하는 높은 성능과 실시간성을 제공하기 어렵다.

1.3. 연구 목표

본 연구에서는 하드웨어 기반의 메모리 보호 기술을 이용한 효과적이고도 안전한 JIT 컴파일된 코드 보호 기법을 제안한다. 우리는 Intel 사의 MPK를 이용하되, 코드스페이스에 할당되는 보호 키를 다양화하여 세밀하고 강력한 보호를 제공한다. 또한 빠르고 안전한 보호 키 할당 정책을 이용하여 메모리 권한 관리의 최소 크기인 4kB 단위로 보호를 제공한다.

2. 연구 배경

2.1. JIT Compile

JIT (Just-in-Time) 컴파일은 프로그램을 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, 프로그램 실행 중에 필요한 부분만 실시간으로 컴파일한다. JIT 컴파일은 코드 최적화와 실행 속도 면에서 큰 이점을 제공한다.

인터프리터 방식은 최적화 과정 없이 번역하므로 같은 코드 입력에 대해서 매번 새로 코드를 번역하기 때문에 비효율적이다. 정적 컴파일 방식은 실행 전 컴파일을 해야 하고 사용하지 않는 코드 또한 전부 번역해야 하기 때문에 시간이 오래 걸린다. 이에 반해 JIT 컴파일러는 전체 코드 중 필요한 부분만 변환하고, 기계어로 변환된 코드는 캐시에 저장하고 다시 컴파일할 필요 없이 재사용하기 때문에 효율적이다.

2.2. Intel MPK

Intel Memory Protection Keys(MPK)는 메모리 접근 제어를 위한 하드웨어 기반의 메모

리 보호 기술이다.[5] MPK는 사용자 공간에서 메모리 페이지 그룹에 대한 접근 권한을 동적으로 관리할 수 있으며, 페이지 테이블 엔트리(PTE)에 보호 키를 할당하고 해당 키의 접근 권한을 설정함으로써 메모리 접근을 제어한다.

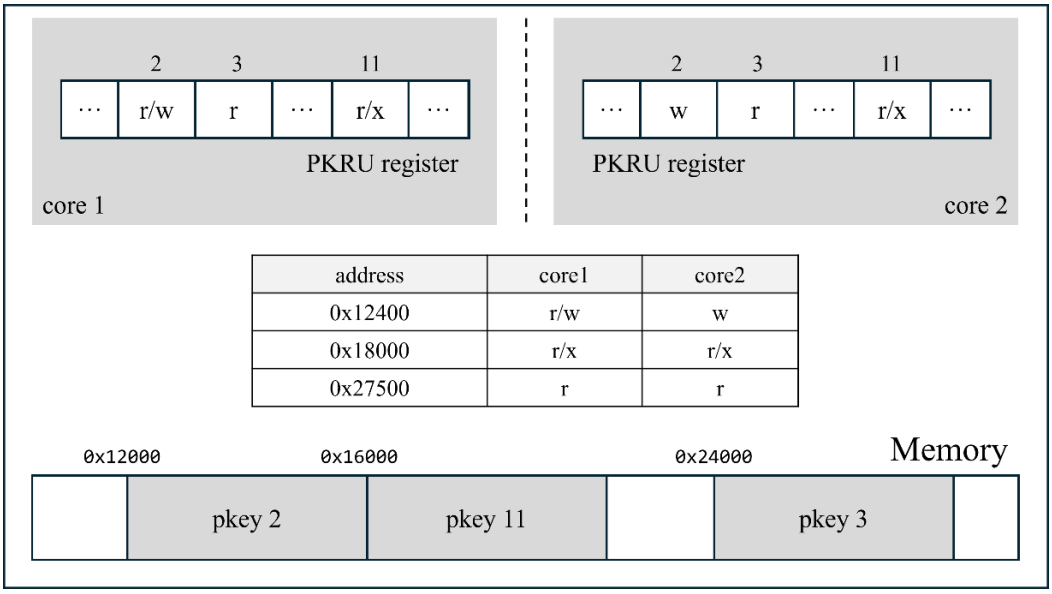

Fig 1. How Intel MPK works

MPK는 PTE의 32~35번째 비트를 사용하여 보호 키 값을 저장한다. 각 메모리 페이지는 4개의 비트를 사용하여 최대 16개의 보호 키를 적용할 수 있다. 또한 보호 키의 접근 권한은 PKRU 레지스터로 관리된다. 각 키의 읽기/쓰기(read/write), 읽기 전용(read only), 접근 불가(no access)와 같은 권한을 2개의 비트를 통해 설정한다. PKRU는 코어별로 관리되어, 각 코어가 독립적으로 다른 메모리 접근 권한을 가질 수 있다. MPK는 PKRU의 값을 제어하기 위해 두 가지 명령어를 제공한다. `WRPKRU` 명령어는 PKRU 값을 업데이트하며, `RDPKRU` 명령어는 현재 PKRU 값을 조회하는 기능을 수행한다.

MPK의 주요 장점은 최대 16개의 서로 다른 페이지 그룹에 대한 접근 권한을 동시에 관리할 수 있다는 점이다. 또한 한 번 보호 키를 할당한 이후에는 PTE를 직접 수정하지 않고도 메모리 접근 권한을 변경할 수 있어 성능상의 이점이 있다.

3. 연구 내용

3.1. Intel MPK의 사용

MPK는 페이지 테이블 항목 자체를 수정하는 대신 PKRU 레지스터의 값만을 바꿔 메

모리 접근 권한을 변경할 수 있어 동적 메모리 보호 시에 성능 면에서 유리하다. JIT 환경에서는 동적으로 생성된 코드를 실행하는 동안 쓰기 권한을 부여해야 하기 때문에 우리는 Intel MPK를 이용해 코드스페이스를 보호한다.

또한 각 PTE에 대해 매핑되는 메모리 크기는 4kB로 한 번에 JIT 컴파일되는 코드 크기에 비하면 매우 크다. 따라서 우리는 한 번에 연속된 4kB보다 큰 메모리가 쓰기 권한을 가지지 않도록 보장하여 최대한 세밀한 보호를 제공한다.

3.2. 코드스페이스 할당

JIT 컴파일된 코드를 저장하기 위해 할당된 4kB 크기의 메모리 블록을 코드스페이스라고 한다. 만약 JIT 컴파일된 코드가 코드스페이스 경계를 넘어 여러 페이지에 걸쳐 저장되면, 코드를 저장할 때 여러 페이지에 동시에 쓰기 권한을 부여해야 하므로 보안 취약점이 발생할 수 있다.

따라서 JIT 컴파일된 코드를 효율적으로 관리하기 위해, 새로운 코드스페이스를 할당할 때 시작 주소와 크기가 4kB에 맞춰진 큰 코드스페이스를 미리 할당하고, 코드스페이스에 순차적으로 코드를 저장하는 방식을 사용한다. 코드스페이스를 미리 정렬하여 할당하는 이유는 페이지 경계를 넘지 않아 하나의 보호 키로만 권한 관리를 할 수 있도록 하기 위해서이다. 이로 인해 코드를 저장할 때 여러 페이지에 동시에 쓰기 권한을 부여하지 않아도 되며, 각 코드스페이스는 독립적으로 보호된다. 우리는 이러한 시작 주소 정렬을 런타임의 내장 할당자를 수정하여 달성하였다.

코드스페이스에 남은 공간이 새로 컴파일된 코드를 저장하기에 충분하지 않다면, 새로운 코드스페이스를 할당하여 나머지 코드를 저장한다. 이를 통해 각 코드스페이스는 분리된 메모리 블록으로 유지되며, 서로 다른 보호 키로 보호된다.

3.3. 보호 키 할당

코드스페이스가 할당되고 나면, 해당 코드스페이스에 접근 권한을 관리할 보호 키를 할당한다. 이를 위해 pkey_gen 함수를 구현하였다. 보호 키를 할당한 방법은 아래와 같다.

3.3.1. 보호 키 설계 조건

연속된 코드스페이스에 대해 서로 다른 보호 키 할당.

우리가 목표하는 세밀한 메모리 보호를 달성하기 위해서는, 인접한 두 개의 코드스페

이스에 대해 서로 다른 보호 키를 할당해야 한다. 만약 인접한 코드스페이스에 동일한 key가 할당되면, 8kB 이상의 메모리 구간에 대해 쓰기가 가능하게 되므로 보안성이 약해질 수 있다. 이를 방지하기 위해 인접한 두 코드스페이스에 서로 다른 보호 키가 할당되도록 설계한다.

예측 불가능한 보호 키.

공격자가 보호 키를 예측할 수 있다면, 하나의 코드스페이스에 대해 컴파일이 일어나고 있는 도중 같은 보호 키를 사용하는 서로 다른 코드스페이스에 대해서도 접근할 수 있게 된다. 이를 방지하기 위해, 공격자가 예측하기 어려운 보호 키 할당 알고리즘을 설계한다.

3.3.2. 코드스페이스와 보호 키 매핑 방법

보호 키 할당을 효율적으로 수행하기 위해 각 코드스페이스의 시작 주소와 보호 키를 매핑하였다. 시작 주소와 보호 키 간의 연결을 저장하는 데에는 두 가지 방법이 있다.

a. 시작 주소와 그에 따른 보호 키를 저장한 테이블 구성

코드스페이스의 시작 주소와 그에 대응하는 보호 키를 테이블에 저장해 두고, 필요할 때마다 이 테이블을 조회하여 해당 키를 확인하는 방식이다. 그러나 이 방식은 테이블을 지속적으로 관리해야 하며, 테이블 자체가 또 다른 보호 대상이 된다. 또한 두 번째 설계 조건을 충족하기 위해서는 보호 키가 예측 불가능해야 하는데, 이 가정 하에서 첫 번째 설계 조건을 충족하기 위한 방법으로는 새로운 코드스페이스에 보호 키를 할당할 때마다 인접한 코드스페이스가 있는지 여부와 해당 코드스페이스의 보호 키 값을 조회하는 수밖에 없다. 그러나 이렇듯 매번 테이블을 조회하는 것은 성능 측면에서 비효율적이다.

b. 시작 주소에 대해 보호 키를 계산하여 사용

테이블을 사용하는 대신, 인접한 코드스페이스에 대해 서로 다른 보호 키를 할당하는 알고리즘을 이용해 필요할 때마다 코드스페이스의 시작 주소로부터 직접 보호 키를 계산하는 방법을 사용할 수 있다. 이 방법은 테이블을 관리할 필요가 없으며, 키 할당 과정이 더 간단하고 효율적이다.

이에 우리는 b 방식을 채택하여 사용하였다.

3.3.3. 보호 키 할당 알고리즘

보호 키 할당 알고리즘은 다음과 같다.

먼저 코드스페이스의 시작 주소에 대해 12만큼 right shift (>>) 연산을 한다. 이는 모든 코드스페이스가 4kB 정렬되어 있어 하위 12비트가 0으로 동일하기 때문이다. 그리고 연산 결과에 대해 해시 연산을 수행한 다음, 해시 연산의 결괏값을 15에 대해 나머지 연산하고 이에 1을 더해 보호 키 값을 확정한다. 이렇게 만들어진 보호 키 값은 1 이상 15 이하로 결정되며, 이는 Intel 사의 아키텍처에서 지원하는 보호 키 중 미리 예약되어 사용되는 0번을 제외한 나머지에 해당한다.[8] 해시 함수는 입력 데이터에 아주 작은 변화가 발생하더라도 출력이 크게 변하도록 설계되어 있고, 특히 인접한 값에 대해 서로 다른 출력을 생성하도록 설계되어 있으므로 연속된 코드스페이스에 대해 서로 다른 계산 결과가 도출될 가능성이 높다.

또한 보호 키의 무작위성을 높이기 위해 해시 연산의 입력값에 일정한 오프셋 값을 더하여 해시 연산을 수행한다. 오프셋 값은 Wasm 런타임이 처음 실행될 때마다 무작위 값으로 지정되며, 서로 다른 실행 회차에는 다른 값으로 지정된다. 오프셋 값은 값의 유출을 막기 위해 커널 수준에서 저장되고 호출된다. 이러한 오프셋 값을 통해 런타임을 반복 수행했을 때 같은 코드스페이스 시작 주소에 대해 서로 다른 보호 키가 할당된다고 보장할 수 있다. 또한 오프셋 값을 런타임의 시작부터 끝까지 유지함으로써 해시 함수를 이용한 디자인의 효용성을 보장했다.

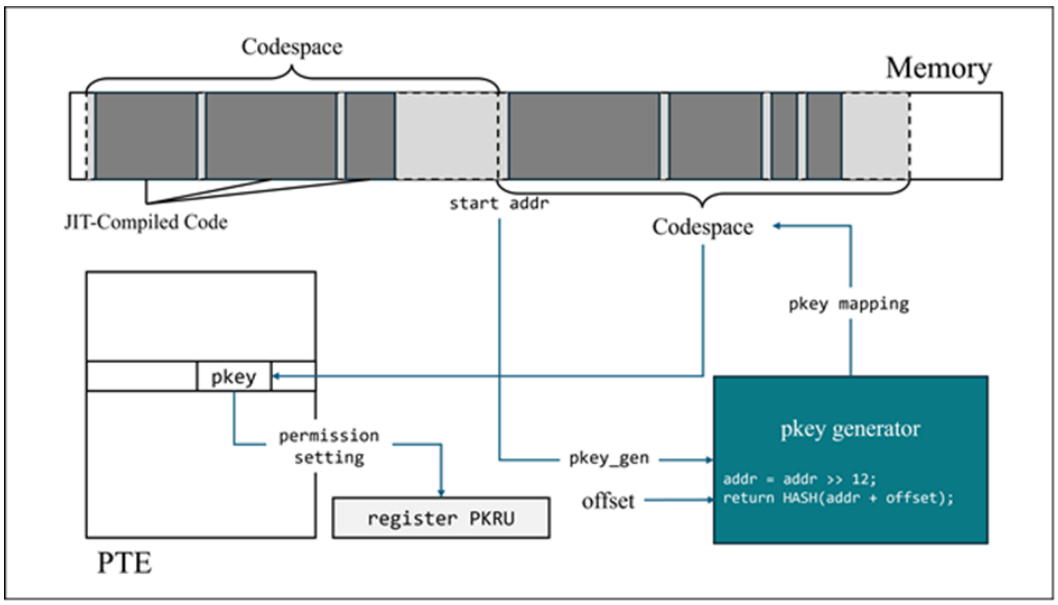

Fig 2. Protecting codespace with Intel MPK

3.3.4. 코드스페이스에 대한 접근 권한 설정

코드스페이스에 대해 보호 키가 할당되고 나면, 코드스페이스의 접근 권한을 적절히 변경해 JIT 컴파일된 코드를 보호해야 한다. 우리는 코드스페이스에 코드를 복사하기 직

전에 쓰기 권한을 부여하고, 코드스페이스에 코드를 모두 쓴 바로 후에 쓰기 권한을 빼앗음으로써 JIT 컴파일된 코드의 변조를 방지했다.

4. 연구 결과 분석 및 평가

우리는 본 연구에 대한 프로토타입을 Wasm Micro Runtime의 fast-jit 위에 구현하고, 이를 대상으로 성능 평가를 진행했다.

4.1. microbench

4.1.1. 랜덤 오프셋 할당

랜덤으로 오프셋을 할당하는 커널 모듈의 적재 여부를 확인하여 이미 적재되어 있다면 해제한 후 다시 적재하고, 적재되어 있지 않다면 적재하여 오프셋 값을 가져오는 함수이다. 해당 함수는 WAMR가 1번 실행되는 동안 1번만 호출되기 때문에 1번 측정하였다. CPU 사이클 수는 121,9051,526이었다.

4.1.2. pkey_set

pkey_set은 linux에서 제공하는 함수로, 보호 키와 접근 권한을 입력으로 받아 보호 키에 대한 접근 권한을 설정한다. pkey_set이 호출되면 입력받은 보호 키에 대응하는 PKRU 비트의 값을 변경해 해당 보호 키가 할당된 메모리 공간에 대한 접근 권한을 변경한다.

pkey_set은 WAMR가 실행되는 동안 여러 번 호출되기 때문에 WAMR가 1번 실행되는 동안 호출된 각각의 pkey_set에 대해 CPU 사이클을 측정하여 평균을 내었다. 초반에는 CPU 사이클 수가 그 이후보다 2배 이상 높게 나왔는데, 이는 캐시 때문인 것으로 추정된다. 따라서 우리는 런타임 실행 초반과 그 이후의 함수 실행에 대해 따로 측정했으며, 초반과 그 이후의 측정 값들이 밀집되어 있다는 것을 증명하기 위해 변동계수를 함께 계산하였다. 초반 CPU 사이클 수는 평균적으로 22,437로 측정되었으며, 변동계수는 약 0.12이다. 이후에는 평균적으로 약 9,888사이클이 나왔으며, 변동계수는 0.12이다.

4.1.3. pkey_gen

pkey_gen 함수는 코드스페이스의 시작 주소를 입력으로 받아 보호 키 값을 생성하고 반환한다. 보호 키의 할당 알고리즘은 4.3 보호 키 할당을 따른다.

pkey_gen 함수 또한 pkey_set과 같이 WAMR가 1번 실행되는 동안 여러 번 호출되기

때문에 pkey_set과 같은 방식으로 CPU 사이클을 측정하였다. 이 역시 처음에 다른 CPU 사이클의 2배 이상의 CPU 사이클이 측정되었는데, pkey_set과 같은 이유로 보인다. 이 역시 함수의 첫 실행과 그 이후의 실행에 대해 따로 측정하고, 변동계수를 계산하였다. 첫 CPU 사이클 수는 24,790이었다. 이후에는 평균적으로 약 10,344사이클로 측정되었으며, 변동계수는 0.11이다.

4.1.4. pkey_mprotect

pkey_mprotect는 linux에서 제공하는 함수로, 메모리 공간의 시작 주소, 크기, 보호 키, 접근 권한을 입력으로 받아 메모리 공간에 보호 키를 할당하고 입력받은 접근 권한으로 해당 PKRU를 초기화한다[9].

해당 함수도 pkey_set과 같은 방식으로 CPU 사이클을 측정하였다. 처음에 측정된 CPU 사이클이 이후에 측정된 사이클의 약 100배였기 때문에, 이 역시 함수의 첫 실행과 그 이후의 실행에 대해 따로 측정하고, 변동계수를 계산하였다. 처음 측정한 CPU 사이클은 3,167,710사이클이었다. 이후에는 평균적으로 30,835사이클로 측정되었으며, 변동계수는 0.21이다.

4.2. 벤치마크 프로그램을 이용한 성능 평가

본 연구에서는 CoreMark, PolyBench 벤치마크 상에서 수정한 런타임의 성능 평가를 진행하였다. CoreMark는 기본적인 읽기/쓰기, 정수 연산, 제어 연산에 중점을 둔 작고 재현 가능한 벤치마크로, 그 단순성과 재현성 덕분에 프로세서의 해김 기능에 대한 성능 오버헤드를 명확하고 안정적으로 평가할 수 있다[10][11]. WAMR의 성능 평가에 있어서는 CoreMark가 널리 사용되고 있으며, 이를 통해 WAMR의 성능을 보다 신뢰성 있게 측정할 수 있다[12]. PolyBench는 선형대수 계산, 동적 프로그래밍 등 다양한 응용 분야에서 추출한 30개의 수치 계산을 수행하는 벤치마크이다[13]. CoreMark가 하나의 프로그램을 벤치마크하는데 반해, PolyBench는 30개의 벤치마크를 수행하기 때문에 보다 다양한 연산에 대해 검증해볼 수 있다.

4.2.1. CoreMark

Parameter	WAMR (No Memory Protection)	WAMR (With Memory Protection)
Total Time (secs)	12.349	12.156
Iterations / Sec	32391.287	32905.561

Table 1. CoreMark Benchmark Test Results

CoreMark 벤치마크를 사용해 기존 WAMR와 메모리 보호 기법을 적용한 수정한 WAMR의 성능을 비교했다. 각각 한 번씩 실행하여 얻은 결과를 비교하였다. Iteration은 400,000회 수행하였고, CoreMark Size는 666단어(2,664바이트)이다.

4.2.2. PolyBench

Parameter	Compared with WAMR (No Memory Protection)		Compared with WAMR (Only Allocator Modified)
	Only Allocator Modified	With Memory Protection	With Memory Protection
2mm	-4%	-3%	1%
3mm	-4%	1%	5%
adi	-4%	-1%	3%
atax	-32%	-32%	0%
bicg	-33%	-33%	0%
cholesky	-8%	-6%	2%
correlation	-10%	-7%	3%
covariance	-7%	-3%	4%
deriche	-18%	-15%	3%

doitgen	-8%	-10%	-2%
durbin	-30%	-30%	0%
fdtd-2d	-4%	-4%	0%
floyd-warshall	-4%	-1%	3%
gemm	-8%	-7%	1%
gemver	-35%	-33%	4%
gesummv	-32%	-34%	-4%
gramschmidt	-3%	-2%	1%
heat-3d	-1%	-1%	0%
jacobi-1d	-28%	-33%	-8%
jacobi-2d	-6%	-5%	0%
ludcmp	-4%	0%	4%
lu	-1%	1%	2%
mvt	-28%	-28%	0%
nussinov	-5%	-5%	0%
seidel-2d	-2%	-3%	-1%
symm	-9%	-2%	7%
syr2k	-10%	-9%	1%
syrk	-10%	-11%	-1%
trisolv	-30%	-30%	0%
trmm	0%	1%	1%

Table 2. PolyBench Benchmark Test Results

Table 2는 PolyBench를 이용해 본 연구의 성능을 평가한 결과를 나타낸다. 표 2에서 볼 수 있듯이, 첫 번째 항목은 PolyBench에서 기존 WAMR의 결과 대비 기존 WAMR에서 할당자만 수정한 WAMR와 최종적으로 구현한 WAMR에서의 오버헤드이다. 두 번째 항목은 기존 WAMR에서 할당자만 수정한 WAMR 대비 최종적으로 구현한 WAMR에서의 오버헤드이다.

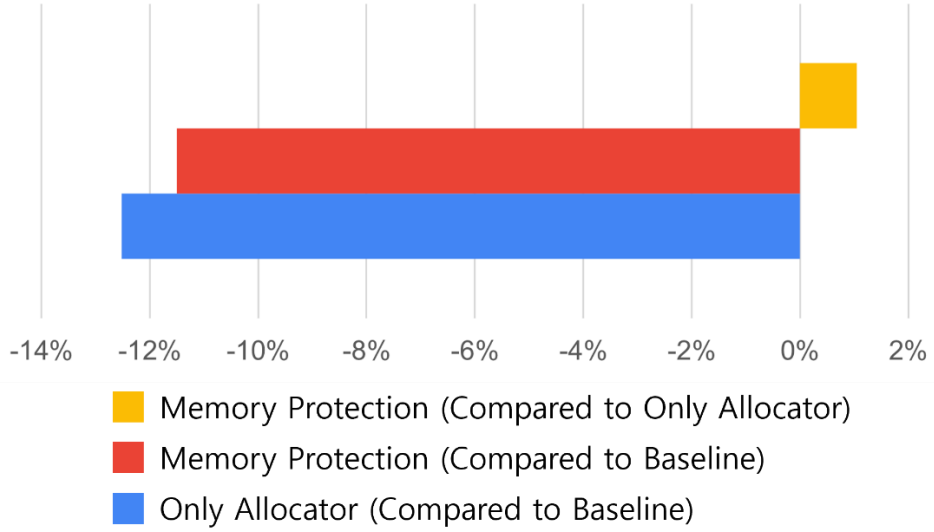

Fig 3. Average Overheads

Fig 3은 PolyBench로 측정한 평균 오버헤드를 나타낸 것이다. 기존 WAMR 대비 할당자만 수정한 WAMR의 평균 오버헤드는 약 13% 감소했으며, 우리가 최종적으로 구현한 WAMR의 평균 오버헤드는 약 12% 감소했다. 또한, 할당자만 수정한 WAMR 대비 최종적으로 구현한 WAMR의 평균 오버헤드는 약 1% 증가했다. 실험 결과에서 알 수 있듯이, 할당자만 수정한 경우와 우리가 최종적으로 구현한 WAMR는 기존 WAMR 대비 전반적으로 성능이 향상되었고, 할당자만 수정한 WAMR와의 성능 차이도 거의 없음을 확인할 수 있다.

4.2.3. 논의

위의 두 절에서 볼 수 있듯이, 우리 연구의 구현은 기존 런타임보다 빠르게 동작한다. 이는 메모리 보호에 대한 기능이 추가되었으니 오버헤드가 발생할 것이라는 예측과는 반대된다. 우리는 이러한 결과가 할당자의 호출 빈도 때문이라고 분석했다. 우리 연구의 구현에서는 이미 할당된 4kB 공간이 모두 사용될 때까지 새로운 공간을 할당할 필요가 없다. 그러나 기존 런타임은 코드 조각이 생성될 때마다 새로운 공간을 할당해야 한다. 이로 인해 우리 연구보다 기존 런타임의 동작이 느려질 수 있다.

Table 2의 3열과 같이, 우리는 MPK를 이용한 보호 없이 할당자만 수정한 프로토타입을 구현하여 우리 연구의 구현과 성능을 비교했다. 그 결과 할당자만 수정한 프로토타입은 기존 런타임보다 평균적으로 약 13% 성능 향상을 보였으며, 우리 연구의 구현은 할당자만 수정한 프로토타입보다 약 1.9% 느렸다. 이 오버헤드는 MPK를 이용해 메모리를 보호하거나 코드스페이스의 주소를 이용해 보호 키를 생성하는 동작 때문에 발생한다. Fig 3에서는 세 경우의 수행 시간 측정치의 평균을 나타내 이 경우에 대하여 직관적으로 설명한다.

5. 결론 및 향후 연구 방향

본 연구에서는 Intel MPK 기반의 메모리 보호 기법을 이용하여 JIT 컴파일된 Wasm 코드를 효과적으로 보호하는 방법을 제안하였다. 구체적으로, 코드스페이스에 할당하는 보호 키를 다양화하였으며, 보안을 달성하기 위해 필수적인 보호 키의 요구사항을 정리하고 이를 만족하는 무작위 키 할당 알고리즘을 만들었다. 또한 이를 Wasm Micro Runtime 상에서 구현하여 프로토타입을 만들었다. 성능 평가 결과에서 보호 기법이 적용되지 않은 WAMR의 런타임보다 최종적으로 구현한 WAMR가 더 빠른 수행 시간을 가지는 것을 확인할 수 있었다. CoreMark를 통한 성능 평가 결과에서는 수정한 WAMR가 기존 WAMR 대비 약 1.59% 성능이 향상되었으며, PolyBench를 통한 성능 평가 결과에서는 우리가 최종적으로 구현한 WAMR의 평균 오버헤드는 약 12% 감소했다. 이는 예상과는 다른 수치로, 할당자의 호출 횟수가 감소함에 따른 것으로 보인다. 이를 검증하기 위해 우리는 메모리 보호 기능을 빼고 할당자만을 수정한 런타임 프로토타입을 구현하여 성능을 측정하였고, 그 결과 기존 WAMR 대비 할당자만 수정한 WAMR의 평균 오버헤드는 약 13% 감소했다. 할당자 호출 횟수를 같게 한 런타임과 비교하여 저하되는 성능 또한 평균 약 5%로 무시할 수 있을 정도로 충분히 낮다. 이를 통해 JIT 컴파일 환경에서 더욱 안전한 웹 애플리케이션 실행이 가능해졌으며, 향후 Wasm의 보안 강화에 기여할 수 있는 중요한 기초를 마련하였다.

6. 참고 문헌

[1] WebAssembly, "WebAssembly: Bringing the web up to speed," https://webassembly.org, (visited on 2024-10-01)

[2] W. De Groef, N. Nikiforakis, Y. Younan, and F. Piessens, "JITSec: Just-in-time Security for Code Injection Attacks," IBBT-DistriNet, Katholieke Universiteit Leuven, Leuven, Belgium.

[3] V8 Documentation. https://v8.dev/docs. Accessed: July 05, 2024

[4] JavaScriptCore. https://developer.apple.com/documentation/javascriptcore. Accessed: July 05, 2024

[5] Intel Corporation, Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B, 3C, & 3D): System Programming Guide, Order No. 325384-084US, June 2024, pp. 136.

[6] Pomonis, M., Petsios, T., Keromytis, A. D., Polychronakis, M., & Kemerlis, V. P. (2017, April). kR^ X: Comprehensive kernel protection against just-in-time code reuse. In Proceedings of the Twelfth European Conference on Computer Systems (pp. 420-436).

[7] Park, T., Dhondt, K., Gens, D., Na, Y., Volckaert, S., & Franz, M. (2020). NOJITSU: Locking down JavaScript engines. In Network and Distributed System Security Symposium. Retrieved from https://www.ndss-symposium.org/ndss-paper/nojitsu-locking-down-javascript-engines/

[8] Pkeys(7) linux programmer's manual, 2018. http://man7.org/linux/man-pages/man7/pkeys.7.html.

[9] Linux Programmer's Manual, "pkey_mprotect," http://man.he.net/man2/pkey_mprotect, (visited on 2024-10-01)

[10] "EEMBC's CoreMark, https://github.com/eembc/coremark," 2022.

[11] M. Bai, R. Pan, and G. Parmer, "OmniWasm: Efficient, Granular Fault Isolation and Control-Flow Integrity for Arm Microcontrollers," in Proc. 2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS), Apr. 2024, pp. 9.

[12] "Performance", github, Oct. 8, 2021, https://github.com/bytecodealliance/wasm-micro-runtime/wiki/Performance, (visited on 2024-09-27)

[13] "Polybench", "https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1", (visited on 2024-09-29)