Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.31, NO.4, Aug. 2021 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2021.31.4.753

WACFI: 웹 어셈블리에서의 간접호출 명령어 보호를 위한 코드 계측 기술*

장 윤 수,[1†] 김 영 주,[2] 권 동 현[3‡]
[1,2,3]부산대학교 (학생, 대학원생, 교수)

WACFI: Code Instrumentation Technique for Protection of Indirect Call in WebAssembly*

Yoonsoo Chang,[1†] Youngju Kim,[2] Donghyun Kwon[3‡]
[1,2,3]Pusan National University (Under graduate student, Graduated student, Professor)

요 약

웹 어셈블리는 웹 환경에서 수행 가능한 명령어 형식을 일컫는다. 최근 웹 어셈블리의 성능적인 우수함 때문에 다양한 웹 애플리케이션에서 웹 어셈블리가 활용되고 있다. 하지만 본 논문에서는 보안 관점에서 웹 어셈블리의 간접호출 명령어에 대한 보호 기능에 취약한 부분이 있다는 것을 알게 되었고, 이에 이러한 웹 어셈블리에서의 간접호출 명령어의 보호를 위한 코드 계측 기술인 WACFI를 제안한다. 구체적으로 WACFI에서는 소스 코드 분석을 통해 얻은 정보를 활용해 웹 어셈블리 코드를 수정하여 웹 어셈블리의 간접호출 명령어 보호 기능을 강화하였다. 우리의 실험 결과에 따르면 WACFI는 단지 약 2.75%의 성능 부하만으로 이러한 보안 기능을 제공하는 것으로 확인되었다.

ABSTRACT

WebAssembly(WASM) is a low-level instruction format that can be run in a web environment. Since WASM has a excellent performance, various web applications use webassembly. However, according to our security analysis WASM has a security pitfall related to control flow integrity (CFI) for indirect calls. To address the problem in this paper we propose a new code instrumentation scheme to protect indirect calls, named WACFI. Specifically WACFI enhances a CFI technique for indirect call in WASM based on source code anlysis and binary instrumentation. To test the feasibility of WACFI, we applied WACFI to a sound-encoding application. According to our experimental results WACFI only adds 2.75% overhead on the execution time while protecting indirect calls safely.

Keywords: WebAssembly, Control Flow Integrity

I. 서 론

웹 어셈블리(web assembly, WASM) [1]는 웹 브라우저와 같은 웹 환경(web environment)에서 수행하는 스택 기반 이진 명령어 형식이다. 개발자는 C/C++, Rust와 같은 프로그래밍 언어로 작성된 코드를 웹 어셈블리 형식으로 크로스 컴파일 함으로써 해당 프로그램을 다양한 웹 환경에서 수행할 수 있다[2]. 심지어 연산 속도 측면에서도 웹 어셈블리 프로그램이 기존의 웹 환경에서 널리 사용되는 자바스크립트로 작성된 프로그램보다 빠른 성능을 보여주기도 하였다[3]. 따라서 앞으로 점차 많은 웹

Received(05. 26. 2021), Modified(06. 10. 2021), Accepted(06. 11. 2021)
* 본 연구는 2020학년도 부산대학교 BK21 FOUR 대학원 혁신지원사업 지원과 부산대학교 기본연구지원사업(2년)에 의하여 연구되었음.
† 주저자, ysc9606@gmail.com
‡ 교신저자, kwondh@pusan.ac.kr(Corresponding author)

응용 프로그램에서 웹 어셈블리를 활용할 것으로 기대된다. 이미 사용자들이 이용 중인 웹 브라우저들 가운데 90% 이상이 웹 어셈블리를 지원하고 있다[4].

한편, 웹 어셈블리는 보안 측면에서도 다양한 기능을 제공하고 있다. 그중 공격자가 간접호출 명령어를 악용해 임의의 함수를 호출하는 공격을 막기 위한 기능이 탑재되었다. 구체적으로 함수 전달 인자와 반환 인자의 타입 정보로 정의된 함수 서명(function signature)을 기반으로 한 제어 흐름 무결성(control flow integrity, CFI) 기능을 제공한다[5].

그러나 이러한 웹 어셈블리의 간접호출 명령어에 대한 보호 기법은 웹 어셈블리의 제한적인 데이터 타입으로 인해 한계를 가지고 있다. 즉 소스 코드상의 다양한 데이터 타입들이 웹 어셈블리의 4가지 타입으로만 다대일 대응이 되고, 이로 인해 소스 코드상에서는 서로 다른 함수 서명이더라도 웹 어셈블리에서는 같은 함수 서명을 가질 수 있게 된다. 이는 결과적으로 웹 어셈블리에서 간접호출 시 함수 서명 검사에서 실제 서로 다른 함수 서명을 갖는 함수들을 구분하지 못하는 상황을 발생시키므로 여전히 코드 재사용 공격(code reuse attack, CRA)으로부터 취약해질 수 있다.

따라서 본 논문은 이러한 웹 어셈블리의 간접호출 명령어로 인한 취약점을 보완하는 기술인 WACFI를 제안한다. WACFI는 기존 웹 어셈블리보다 구체적인 함수 서명을 기반으로 제어 흐름 무결성 검증 기술을 제공한다. 이를 위해 WACFI는 소스 코드 분석기술과 웹 어셈블리 코드 수정기술을 활용하여 구현되었다. 실험 결과 이러한 코드 수정에도 불구하고 WACFI는 기존 웹 어셈블리 코드에 2.75%만의 적은 성능 부하는 일으키는 것으로 확인되었다.

II. 배경 지식

2.1 코드 재사용 공격

코드 재사용 공격은 공격자가 주소 공간에 악의적인 코드를 삽입하지 않고 프로세스의 제어 흐름을 변조하여 기존 코드들을 재사용하는 공격 기술이다[6]. 대표적으로 반환 지향 프로그래밍(return-oriented programming, ROP) 및 점프 지향 프로그래밍(jump-oriented programming, JOP)가 있다. 이러한 공격에서 공격자는 제어 흐름 탈취를 위해 필수적으로 메모리 취약점을 악용하여 간접 분기(indirect branch)의 동작을 변조하게 된다. 구체적으로 간접 분기의 타겟 주소가 저장된 스택이나 레지스터의 값을 변조하여 프로세서의 원래 의도된 제어 흐름이 아닌 공격자가 의도한 제어 흐름을 변경하는 것이다. 대표적인 간접 분기 명령어로 C언어의 함수 포인터 등을 표현할 때 사용되는 간접호출 명령어가 있다.

2.2 웹 어셈블리

웹 어셈블리는 웹 수행 환경에서 수행하는 명령어 스택 기반 바이너리 형식을 일컫는다. 다음은 이러한 웹 어셈블리의 세부 기술 중 본 논문과 관련된 일부분을 다루도록 하겠다.

2.2.1 웹 어셈블리 섹션과 데이터 타입

웹 어셈블리 코드는 다른 바이너리 형식과 유사하게 여러 개의 섹션들로 이루어져 있다. **코드 섹션**의 경우 작성된 개별 함수의 구현을 담고 있으며, **함수 섹션**은 주어진 함수 인덱스(func_idx)에 따라 해당하는 함수들의 코드 섹션 내 주소를 담고 있는 테이블이다. **테이블 섹션**에는 테이블 인덱스(tbl_idx)에 따른 함수 인덱스를 매핑하는 함수 테이블(function table)이 있는 섹션이다. 간접호출 시에 이 테이블 섹션을 참조하게 된다. **타입 섹션**에는 현재 웹 어셈블리 코드에 존재하는 함수들의 서명 정보를 담고 있다. 이때 함수 서명은 함수의 전달 인자들과 반환 값의 타입을 이용해 정의된다.

웹 어셈블리의 데이터 타입은 4가지이다. 예를 들어, i32와 i64는 각각 32비트와 64비트의 정수를 위한 타입이고, f32와 f64는 각각 32비트, 64비트의 실수를 위한 타입을 나타낸다.

2.2.2 웹 어셈블리의 함수 호출

웹 어셈블리는 직접 호출(direct call)과 간접 호출(indirect call) 방식의 함수 호출 명령어들을 제공한다.

직접 호출을 위한 웹 어셈블리의 명령어 이름은 call이며 피연산자로 함수 인덱스가 상수로 명세 되어 있다. 즉 이러한 함수 인덱스는 함수 섹션을 참조하는 데 활용되어 특정 함수의 주소만을 얻을 수 있으므로 이를 통해 calll 명령어를 통해서는 특정 함

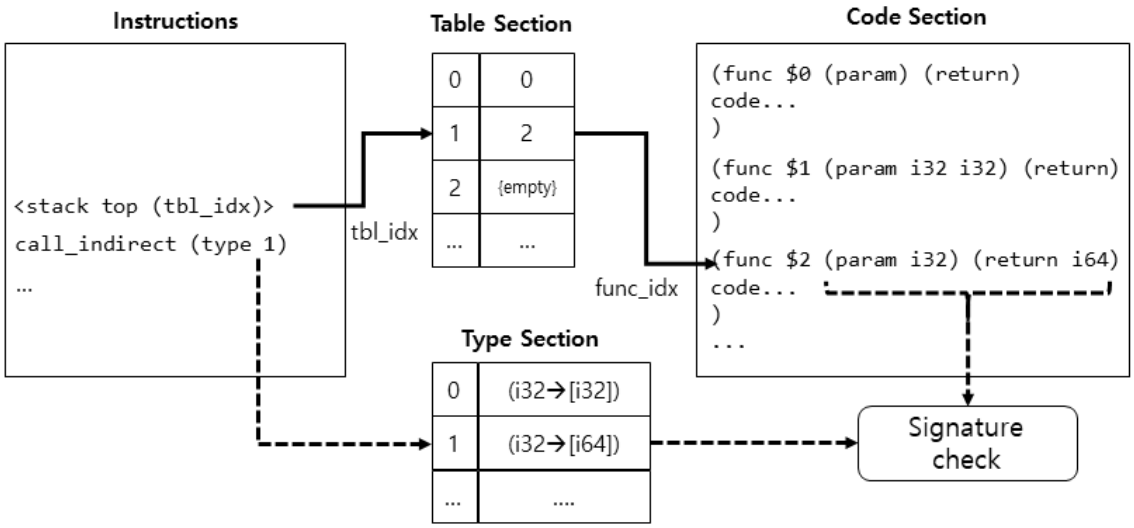


Fig. 1. Process of Indirect Call Instruction of WebAssembly

수만을 호출할 수 있게 된다.

한편 웹 어셈블리는 간접호출을 위해서 call_indirect이라는 간접호출 명령어를 제공한다. call 명령어와는 다르게 call_indirect 명령어로 호출될 함수에 대한 정보를 명령어의 피연산자가 아닌 스택 최상위 요소의 값을 활용하게 된다. 즉 하나의 call_indirect 명령어라 할지라도 수행 직전 스택 최상위 요소의 값이 다르면 서로 다른 함수를 호출할 수 있다. 또한, call_indirect를 호출하기 전 스택의 최상위 요소는 함수 인덱스가 아닌 테이블 인덱스가 저장되어야 한다. 이후 call_indirect가 호출되면 해당 테이블 인덱스를 활용해 테이블 섹션 내의 함수 테이블 참조하여 함수 인덱스를 얻는다. 그리고 이 함수 인덱스를 이용하여 수행할 함수를 결정하게 된다. 한편 call_indirect 명령어는 피연산자로 타입 인덱스(type_idx)라는 상숫값을 가지게 되는데 이 값은 타입 섹션에 정의된 함수 서명 정보를 가져오는 데 사용된다. 즉, 마지막으로 함수 인덱스를 통해 결정된 함수의 서명이 타입 인덱스를 통해 얻은 서명과 일치하는지를 확인하여 일치할 경우에만 수행하고 불일치할 경우 런타임 에러를 발생시키게 된다. "Fig.1. "은 이러한 call_indirect 명령어의 동작을 도식화한 것이다. 예를 들어 "Fig.1. "의 call_indirect의 피연산자로 명세된 타입 인덱스는 1이고 이는 하나의 i32 타입의 전달 인자와 i64 타입의 반환 값을 가지는 함수의 서명 정보(i.e., (i32->(i64))[1])를 나타낸다. 그리고 함수 인덱스에 의해 특정된 $2 함수의 경우에 바로 타입 인덱스로 정의된 서명과 동일한 서명을 지니기 때문에 $2 함수의 수행이 가능한 경우를 보여준다.

III. 가정하는 위협 모델

우리는 다음과 같은 공격모델과 가정들을 토대로 WACFI를 설계하였다. 기본적으로 웹 어셈블리가 제공하는 보안 기능[1]을 통해 방어되는 코드 삽입 공격(code injection attack)이나 반환 명령어 기반의 코드 재사용 공격이 아닌 간접호출을 이용한 코드 재사용 공격을 방어 대상으로 한다. 대신 우리는 기존 연구[5]와 마찬가지로 공격자가 웹 어셈블리의 다양한 메모리 취약점을 활용할 경우, 간접호출 명령어가 사용할 테이블 인덱스를 임의로 조작하여 스택에 저장할 수 있다고 가정한다. 결과적으로 이를 통해 공격자는 제어 흐름을 조작해 스크립트 실행 함수와 같은 중요 함수를 악의적으로 호출하거나 함수 재사용 공격(function reuse attack) [7]과 같은 코드 재사용 공격을 일으킬 수 있다.

1) '->' 기호를 기준으로 왼쪽에 나열되는 데이터 타입들은 함수 파라미터들의 데이터 타입이고, 오른쪽은 반환 값의 데이터 타입이다.

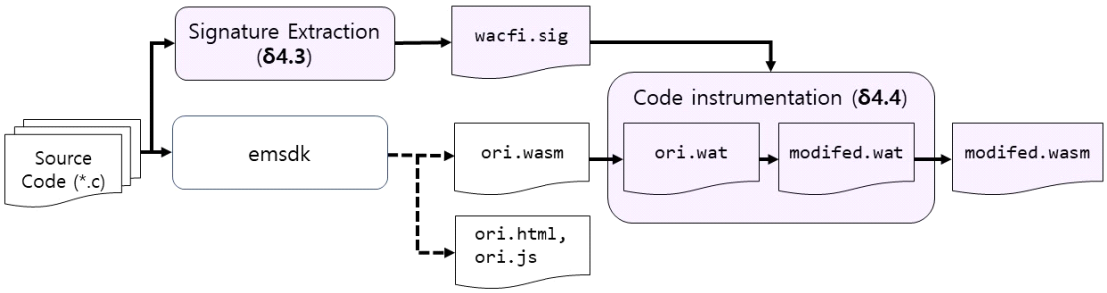


Fig. 2. Process of WACFI. The pink parts are the procedures added by WACFI.

IV. WACFI

4.1 동작 개요

"Fig. 2."는 일반적인 웹 어셈블리 개발 과정과 본 논문에서 제안하는 WACFI에 의해 추가되는 과정들을 보여준다. 먼저 본 연구에서는 Emscripten SDK (emsdk)[8]를 활용하여 웹 어셈블리 코드를 생성하였다. emsdk를 활용하여 소스 코드를 컴파일하게 되면 웹 어셈블리 코드(.wasm)뿐만 아니라 이를 브라우저와 같은 웹 환경에서도 수행할 수 있도록 하는 래퍼 함수(wrapper code)가 들어있는 .html와 .js 코드를 생성해준다. WACFI는 그중에서 웹 어셈블리 코드와 ("Fig.2."의 ori.wasm) 소스 코드를 입력으로 받아 최종적으로 indirect call에 대한 수행 흐름 무결성 보호 기술이 적용된 웹 어셈블리 코드를 ("Fig.2."의 modified. wasm) 생성하게 된다.

이를 위해 WACFI는 크게 두 단계로 동작하게 된다. 먼저 개발자가 작성한 소스 코드를 분석하여 얻은 상세한 데이터 타입 정보를 기반으로 한 새로운 함수 서명 정보를 생성한다(4.3장). 다음으로 생성된 서명 정보를 기반으로 웹 어셈블리 코드의 간접호출 명령어에 수행 흐름 무결성 보호를 위한 코드 계측 단계가 이루어지게 된다(4.4장). 각 단계에서 이루어지는 구체적인 내용은 해당 장에서 설명하도록 하겠다.

4.2 웹 어셈블리의 함수 서명 방식의 문제점

앞서 1장에서 설명한 바와 같이 웹 어셈블리는 간접호출에 대한 함수 서명 기반의 보호 기술을 제공한다[5]. 웹 어셈블리는 반환 값의 타입과 전달 인자들의 순서 및 데이터 타입 정보를 이용해 각 함수 서명 값을 계산한다. 하지만, 웹 어셈블리는 4가지의 데이터 타입만을 가지고 있고 이는 소스 코드에 사용된 C/C++이나 Rust와 같은 언어들보다 훨씬 적은 수이다. 예를 들어, C에서 int, long, char와 같은 32비트 이하의 데이터 타입들은 웹 어셈블리의 i32 타입으로 변환된다. 그러므로 소스 코드들의 데이터 타입을 기준으로는 서로 다른 함수 서명을 가지는 함수들이 웹 어셈블리 코드상에서는 동일한 함수 서명을 가지게 될 수 있다. "Fig.3. "의 (a)에 나와 있는 func1, func3 함수의 경우 이러한 예를 보여준다. C 언어의 데이터 타입을 기준으로 함수의 서명을 계산하면 둘의 반환 타입이 다르므로 (각각 int와 char) 함수 서명도 달라야 한다. 하지만 "Fig.3. "의 (b)에 나와 있듯이 웹 어셈블리에서는 func1와 func3은 동일한 (i32->[i32])라는 함수 서명을 가지게 된다.

이러한 함수 서명 방식의 문제는 결과적으로 공격자가 웹 어셈블리의 함수 서명 방식을 우회하여 적법하지 않은 함수를 호출할 수 있게 한다. 예를 들어, "Fig.3. "의 (a)에 나오는 fn 함수 포인터의 경우 소스 코드의 데이터 타입 정보에 따르면 선언된 함수들 가운데 func1만을 호출하는 것이 적법하다. 하지만 "Fig.3. "의 (b)에서 볼 수 있듯이 웹 어셈블리의 함수 서명 방식에 따르면 func1, func2, func3, func4가 모두 동일한 함수 서명을 가진다. 따라서 공격자는 fn에 해당하는 간접 호출 명령어를 악용해 이러한 함수들을 모두 호출할 수 있게 된다.

4.3 소스 코드 분석 기반 함수 서명 추출

WACFI는 4.2장에서 설명한 웹 어셈블리의 함수 서명 방식의 문제를 개선하기 위해 소스 코드 분석 기반으로 함수 서명을 계산하였다. 이를 위해, 먼저

```
int func1 (int) {
  …
}
short func2 (short) {
  …
}
char func3 (int) {
  int (*fn)(int);
  …
  fn(0);
}
short func4 (short) {
  …
}
```

(a) Example source code

	WASM_sig	WACFI_sig
func1	**(i32→[i32])**	**(i32→[i32])**
func2	**(i32→[i32])**	(i16→[i16])
fn	**(i32→[i32])**	**(i32→[i32])**
func3	**(i32→[i32])**	(i32→[i8])
func4	**(i32→[i32])**	(i16→[i16])

(b) Function Signatures

func_idx	func_ID
1	4
2	2
3	1
4	3

(c)func_ID mapping table

WACFI_sig	l_bnd	u_bnd
(i32→[i8])	1	1
(i16→[i16])	2	3
(i32→[i32])	4	4

(d)WACFI sig bounds table

Fig. 3. Function Signature and Metadata of WACFI

우리는 소스 코드 분석 도구 [9]를 활용하여 주어진 소스 코드 내에 정의된 함수들의 서명정보들을 추출하였다. 예를 들어, "Fig.3. "의 (b)에 나온 WACFI의 함수 서명 결과(WACFI_sig)를 보면 사용되는 데이터 타입이 세분화되어 func1의 함수 서명이 소스코드 상에서의 다른 func2, func3, func4와 구분되어 진 것을 확인할 수 있다. 마찬가지로 우리는 이렇게 얻어진 함수 서명 정보를 토대로 소스 코드에 내에 사용되는 간접 호출 명령어들을 찾아내고 해당 간접 호출 명령어로 호출할 수 있는 함수 서명을 정의하였다. 예를 들어, "Fig.3. "의 fn은 함수포인터를 위한 간접 호출 명령어를 가리키며 func3의 fn 함수포인터 선언부분을 분석함으로써 해당 간접 호출 명령어를 통해 호출할 수 있는 함수 서명은 (i32-〉[i32]) 이라는 것을 보여주고 있다. 결과적으로 이렇게 얻은 각 함수별 새로운 함수 서명 정보와 간접 호출 명령어들의 적법한 함수 서명에 대한 정보는 별도의 파일로(wacfi.sig)에 저장하여 뒤이어 나올 코드 수정 모듈에서 활용 가능하도록 하였다.

4.4 수행 흐름 무결성 보호를 위한 코드 계측

WACFI는 4.3장에서 얻은 정보를 활용해 간접호출 명령어들을 보호하는 기능을 기존의 웹 어셈블리 코드에 추가하기 위해 이러한 웹 어셈블리 코드를 계측하였다. 이 때 emsdk로 컴파일한 결과는 웹 어셈블리 이진 형식인 .wasm("Fig.2. "의 ori.wasm) 이기 때문에 우리는 이를 WABT[10]라고 하는 툴체인의 wasm2wat 도구를 활용하여 .wat이라고 하는 웹 어셈블리 텍스트 형식으로 변환한 파일 상에서 계측을 진행하였다. 계측을 마친 후에는 마찬가지로 WABT의 wat2wasm을 활용하여 WACFI의 보호 기능이 탑재된 웹 어셈블리 이진 형식 파일을 생성하였다. ("Fig.2. "의 modifed.wasm)

구체적으로 WACFI에 코드 계측을 통해 추가한 내용은 다음과 같다. 먼저 우리는 기존의 웹 어셈블리 코드가 사용하는 함수 인덱스마다 새로운 함수 식별자 (func_ID)를 할당하였다. 이 때 함수 식별자는 각 함수 인덱스 하나에 하나씩 대응되며 식별자를 할당하는 방식은 WACFI가 정의한 함수 서명을 기준으로 동일한 서명을 가지는 함수들이 연속적인 값을 갖도록 하였다. WACFI는 이러한 함수 인덱스와 함수 식별자의 매핑관계를 정의한 매핑 테이블(func_ID mapping table)을 웹 어셈블리 코드의 테이블 섹션에 추가하였다. 또한 각각의 WACFI 함수 서명별 함수 식별자 범위를 저장하는 함수 서명 바운드 테이블(WACFI_sig bounds table)을 생성한다. 예를 들어, "Fig.3. "의 (b)에서 func2와 func4는 WACFI의 서명정보 상으로 동일한 함수 서명을 갖는 함수들이다. 그리고 "Fig.3. "의 (c)를 보면 알 수 있듯이 func2와 func4의 함수 인덱스(func_idx)는 각각 2와 4라는 값을 갖지만 함수 식별자(func_ID)에는 2와 3이라는 새로운 값이 할당된다. "Fig.3. "의 (d)는 주어진 예제코드에 대한 테이블을 보여주며 (i16-〉[i16] 함수 서명의 경우 적법한 함수 식별자가 2와 3이므로 lower

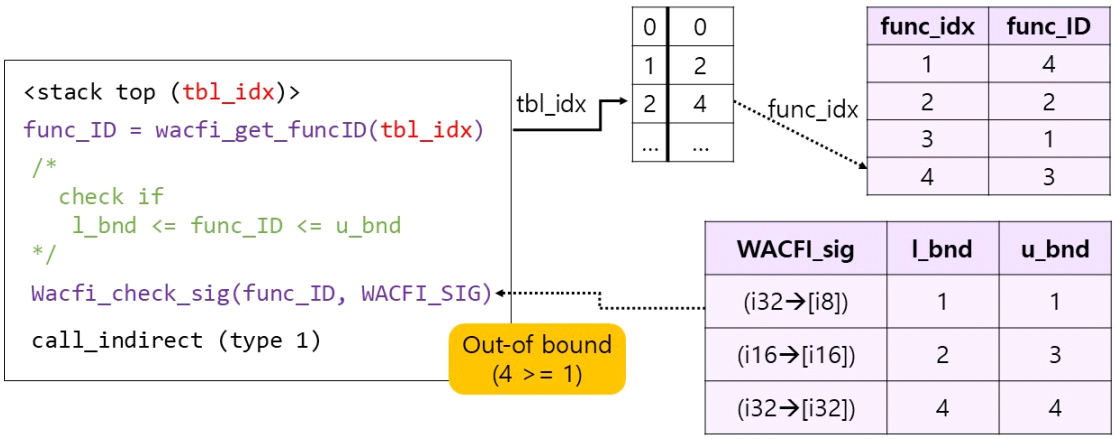


Fig. 4. Instrumentation of WACFI

bound(l_bnd)와 upper bound(u_bnd)가 각각 2와 3인 것을 볼 수 있다. 이러한 식별자의 범위(bound)는 호출된 함수가 갖는 WACFI 함수 서명에 대해 범위 검사를 가능케 할 것이다.

다음으로 우리는 WASM 코드 내에 있는 간접호출 명령어 앞에 WACFI의 서명정보를 토대로 수행흐름 무결성을 보호하는 코드를 추가하였다. 먼저 우리는 스택에 저장된 테이블 인덱스를 가져와 이를 통해 함수 식별자 매핑 테이블을 참조하여 호출하고자 하는 함수 식별자를 얻어낸다. 다음으로 이러한 식별자의 값을 통해 적법성을 검증하게 된다. 이 때 적법성 여부는 4.3장에서 얻은 각 간접호출 마다 호출 가능한 함수 서명(WACFI_sig, "Fig.3."의 (b)에 나타난 것과 동일)과 함수 서명 바운드 테이블을 활용한다. 즉 호출하고자 하는 함수의 함수 식별자가 간접 호출 명령어의 함수 서명에 해당하는 함수 식별자 범위 내에 있는지 확인하는 것이다. 예를 들어, "Fig.4. "는 이러한 검증과정을 적용해 WACFI가 공격자에 의한 테이블 인덱스의 변조 공격을 탐지하는 과정을 보여준다. 즉 변조된 테이블 인덱스를 통한 함수 식별자(현재 예제에서는 '3')는 해당 간접호출 명령어로 호출 가능한 함수 서명(WACFI_SIG_0)의 함수 식별자 범위인 1이상 1이하에 해당하지 않을 것이므로 변조되었음을 확인할 수 있다.

V. 실험 결과

우리는 다음과 같은 실험환경을 가지고 성능을 측정하였다. 먼저 우리는 Intel i5-8265U 프로세서와 기본 주파수 1.60GHz, 8GB의 RAM이 탑재된 머신을 사용하였고 Ubuntu 18.04.5 LTS 64bit을 운영체제로 사용하였다. 웹 어셈블리 코드는 Emscripten SDK 컴파일러 툴체인을 사용하여 생성하였으며 구체적으로 EMCC(Emscripten cc/clang-like replacement + linker emulating GNU ld) 는 2.0.1 버전, clang은 12.0.0 버전, 스레드 모델(Thread model)은 posix로 사용하였다. 테스트 브라우저는 Chrome Canary ver.90.0.4427.5를 이용하였다.

그리고 WACFI 적용시의 성능부하를 측정하기 위한 벤치마크로 Miniaudio[11]의 최신 버전을 사용하였다. Miniaudio는 오디오 캡처와 재생을 위한 C로 작성된 라이브러리이다. 단, "Fig.5."과 같이 Miniaudio는 다양한 기능들을 제공하고 각 기능들에 해당하는 코드가 구분이 되어 있어 이에 각 기능을 구현한 코드 별로 수행 시간을 측정하였다. 이 때 개별 기능을 구현한 함수들을 간접호출 명령어로 호출하도록 되어있어 WACFI에 의해 간접호출 명령어가 보호될 경우 발생하는 동작 시간과 코드 크기 부하를 측정하였다.

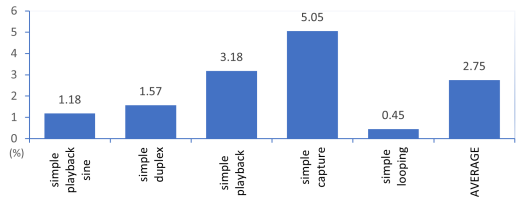


Fig. 5. Performance Time Overhead

5.1 수행 시간 성능 부하

우리는 측정 시에 발생하는 실험 오차로 인한 영향을 줄이기 위해 Miniaudio가 제공하는 개별 기능들을 5,000회 반복하도록 수정하고 그 때 WACFI 적용 전후의 수행 시간을 측정해 성능 부하를 계산하였다. 결과는 "Fig. 5."에서와 같이 2.75 %의 평균 성능 부하가 측정되었다.

5.2 코드 크기 부하

코드 크기 부하는 벤치마크에서 주요 기능을 수행하는 각 파일이 기존(origin)과 WACFI를 적용 후 얼마나 코드 크기가 증가하였는지 측정하였다. 코드 크기는 "Fig.6."와 같이 1% 내외로 증가하였다. 이는 "Fig.4. "에 나오는 wacfi_check_sig의 경우 코드를 인라인 하지 않고 별도의 함수로 정의 한 뒤 개별 간접호출 명령어 앞에는 wacfi_check_sig 함수를 호출해 사용하도록 하여 WACFI로 추가되는 코드 중 중복되는 코드의 양을 줄였기 때문으로 보인다.

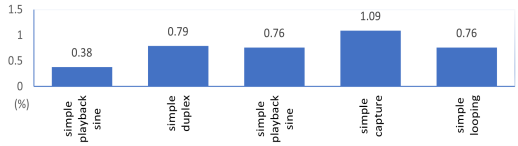


Fig. 6. Code Size Overhead

VI. 관련 연구

6.1 제어 흐름 무결성 연구

제어 흐름 무결성(control flow integrity, CFI)은 간접 호출 명령어를 악용한 CRA를 방어하기 위한 보안 기술 중 하나로 프로그램이 적법한 제어 흐름으로만 수행될 수 있도록 제한하는 기술이다.

이 때 적법한 제어 흐름을 정의하는 방식에 따라서 여러 가지로 나뉘게 되는데 대표적으로 컴파일러 분석 시에 활용되는 제어 흐름 그래프(control flow graph, CFG)를 활용하는 연구들이 있다[12]. 수행 흐름 그래프는 코드의 기본 블록(basic block)을 노드로, 적법한 제어 흐름을 엣지로 나타내는 그래프이다. 즉 이러한 연구들에서는 간접 호출 명령어를 포함한 간접 분기 명령어를 통해 제어 흐름이 변경될 때 수행 흐름 그래프 상에 정의된 엣지들 중 하나로만 변경될 수 있도록 하여 공격자가 제어 흐름을 변조하는 것을 방어한다. 하지만 이러한 방식의 경우 결국 제어 흐름 그래프의 정확도에 따라 보안성이 결정되게 되는데 일반적으로 프로그램에 대한 완전한 제어 흐름 그래프를 생성하는 것은 열린 문제로 알려져 있다[13].

한편 또 다른 방식으로 함수의 서명 정보를 사용하는 방식([14], [15], [16])이 있다. 즉 간접 호출 명령어마다 호출 가능한 함수의 서명 정보를 정의해 두고, 수행 중 간접호출 명령어가 수행하고자 하는 타겟 함수가 동일한 서명 정보를 가지고 있는지 확인하여 공격자가 임의로 간접 호출 명령어를 이용해 적법하지 않은 함수를 호출하는 것을 방지하는 방식이다. 이 때, 함수의 서명 정보는 다양한 방식으로 정의될 수 있는데 [14]의 경우에는 함수의 전달인자 개수와 반환 값의 유무로, WASM의 경우에는 2.2장에서 설명한 바와 같이 WASM에서 제공하는 데이터 타입을 기준으로 전달인자들의 데이터 타입 및 반환 값의 데이터 타입을 가지고 함수의 서명을 정의하였다. 하지만 4.2장에서 설명한 바와 같이 WASM의 제한적인 데이터 타입 시스템으로 인해 WASM의 함수 서명 기반 보호 기술은 결과적으로 보안상 취약점을 가지게 된다[17]. WACFI도 이러한 기존 방식들과 같이 함수의 서명 정보를 활용한 연구의 일환이지만 기존 방식들과는 다르게 소스코드에 개발자가 정의한 상세한 함수의 서명 정보(소스코드의 프로그래밍 언어가 지원하는 데이터 타입을 기준으로 한 전달인자들과 반환 값 데이터 타입)을 활용한다는 점에서 더 나은 제어 흐름 무결성 보호 기능을 제공한다.

한편 WACFI와 마찬가지로 소스코드 상에서 상세한 함수의 서명 정보를 활용한 연구들[15][16] 같은 경우에는 x86과 같은 잘 알려진 머신 코드를 대상으로 수행 흐름 무결성 보호를 위한 코드를 추가하였다. 하지만 이러한 머신에서는 간접 호출 명령어가 레지스터 혹은 메모리 영역에 단순히 저장되어 있는 주소를 활용한다는 점에서 2.2장에서 설명한 WASM에서의 간접 호출 명령어의 동작과 차이가 있다. WACFI는 바로 이러한 WASM의 고유한 간접 호출 명령어 동작과정과 WASM의 명령어들을 활용해 WASM에 특화된 수행 흐름 무결성 검증 기술을 제공한다는 점에서 이러한 기존 연구들과 차별

성을 가진다.

6.2 웹 어셈블리의 보안

웹 어셈블리의 확장성 및 성능상 이점으로 인하여 점차 많은 웹 어플리케이션에 활용이 되면서 한편으로는 이러한 웹 어셈블리 동작의 보안성에 대한 연구들도 활발히 이루어지고 있다. [5],[17]에서는 웹 어셈블리가 가진 정수 오버플로우 취약점, 포맷 스트링 취약점, 버퍼 오버플로우 취약점 등 다양한 취약점들을 예제와 함께 보이고 이를 활용한 공격 기법들에 대해 소개하였다. 특히 [17]에서는 본 논문과 마찬가지로 웹 어셈블리의 간접 호출 명령어가 공격에 악용될 수 있는 문제를 동일하게 제기할 뿐만 아니라 이를 방지할 수 있는 WASM 언어 기반의 대책에 대해서도 설명하고 있다. 즉 WASM에서 함수 테이블을 여러 개 두고, 각 간접 호출 명령어가 동작할 때 참고할 함수 테이블을 지정할 수 있도록 WASM 언어를 확장하자는 제안이 있는데 이를 활용하면 기존 WASM의 간접 호출 명령어 보호 기법을 보완할 수 있다는 것이다. WACFI 역시 WASM의 함수 테이블과 별개로 함수 서명 바운드 테이블을 활용한다는 점에서 해당 제안과 유사하지만, 해당 제안은 단순히 여러 개의 함수 테이블을 지원하기 위한 WASM 언어의 지원일뿐 구체적인 제어 흐름 무결성 보호를 위한 방법론이나 구현은 없다는 점에서 한계가 존재한다. 더구나 이러한 제안은 정식 WASM 언어 표준에는 등록되지 않았을 뿐더러 등록이 된다 하더라도 이를 지원하기 위해서는 모든 WASM을 지원하는 브라우저 등의 수정이 필요한 만큼 실사용까지 상당한 시간이 걸릴 것으로 예상된다. 하지만 WACFI는 현재 WASM 언어로 작성된 바이너리에 바로 적용이 가능하다.

추가적으로 [17]에서는 WASM의 메모리 취약점 방어를 위한 활용 가능한 기술 제안들도 소개하였지만 이 역시 아직 WASM 표준에 들지 못하였을 뿐만 아니라 확률 기반의 보호로 인해 여전히 취약점이 존재할 수 있거나 상당한 성능 부하를 발생시키는 단점이 존재한다. 또한 WACFI는 현재 WASM내의 메모리 취약점을 방어하는 것이 아닌 이를 공격자가 악용하는 상황에서의 제어 흐름 무결성 보호 기술로써 이러한 기술들과 가정하는 위협 모델에서도 차이가 있다.

VII. 결 론

이번 논문에서 우리는 웹 어셈블리의 간접호출 명령어 보호를 위한 코드 계측 기술인 WACFI를 제안하였다. 구체적으로 WACFI는 기존의 웹 어셈블리의 부족한 데이터 타입을 활용하는 대신에 소스 코드에서 추출한 데이터 타입을 기반으로 한 새로운 함수 서명 기반의 무결성 검증 메커니즘을 제안하였다. 우리의 실험 결과는 WACFI가 이러한 보안 기능을 추가함에도 평균 2.75%의 성능 부하를 일으킨다는 것을 보여준다. 향후 소스 코드 분석기술 확장을 통해 C 언어 외에 WASM이 지원하는 다양한 프로그래밍 언어 기반의 프로그램에도 적용하는 방향이나 기존의 다른 머신 코드를 대상으로 한 함수 서명 기반의 수행 흐름 무결성 연구들을[15],[18] 참고하여 보다 정밀한 제어 흐름 무결성 보호 기법으로의 확장이 가능할 것으로 보인다.

References

[1] WebAssembly, https://webassembly.org, Feb. 2021.

[2] Haas, A., and Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Bastien, J. F. "Bringing the web up to speed with WebAssembly." In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 185-200, June. 2017.

[3] devlopers google, https://developers.google.com/web/updates/2019/02/hotpath-with-wasm, Feb. 2021.

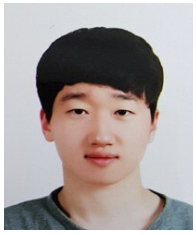
[4] canIuse, https://caniuse.com/?search=WebAssembly, Feb. 2021.

[5] McFadden, B., and Lukasiewicz, T., Dileo, J., Engler, J. "Security chasms of wasm." NCC Group Whitepaper. Aug. 2018.

[6] Göktas, E., and Athanasopoulos, E., Bos, H., Portokalidis, G. "Out of control: Overcoming control-flow integrity." In 2014 IEEE Symposium

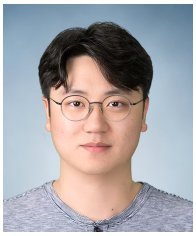
on Security and Privacy pp. 575-589, May. 2014.
〔7〕 Guo, Y., and Chen, L., Shi, G. "Function-oriented programming: A new class of code reuse attack in c applications." In 2018 IEEE Conference on Communications and Network Security pp. 1-9, May. 2018.
〔8〕 Emscripten SDK, https://emscripten.org/, Feb. 2021.
〔9〕 LLVM, https://llvm.org/, Feb. 2021.
〔10〕 wabt, https://github.com/WebAssembly/wabt, Feb. 2021.
〔11〕 miniaudio, https://github.com/mackron/ miniaudio, Feb. 2021.
〔12〕 Abadi, M., and Budiu, M., Erlingsson, U., Ligatti, J. "Control-flow integrity principles, implementations, and applications." ACM Transactions on Information and System Security, vol.13 no.1, pp. 1-40. Oct. 2009.
〔13〕 Checkoway, S., and Davi, L., Dmitrienko, A., Sadeghi, A. R., Shacham, H., Winandy, M. "Return-oriented programming without returns." In Proceedings of the 17th ACM conference on Computer and communications security, pp. 559-572, Oct. 2010.
〔14〕 Van Der Veen, V., and Göktas, E., Contag, M., Pawoloski, A., Chen, X., Rawat, S., Giuffrida, C. "A tough call: Mitigating advanced code-reuse attacks at the binary level." In 2016 IEEE Symposium on Security and Privacy (SP), pp. 934-953, May. 2016.
〔15〕 Pax Team. RAP: RIP ROP. https://pax.grsecurity.net/docs/ Feb. 2021.
〔16〕 Niu, B., and Tan, G. "Modular control-flow integrity." In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 577-587, June. 2014.
〔17〕 Lehmann, D., and Kinder, J., Pradel, M. "Everything old is new again: Binary security of webassembly." In 29th {USENIX} Security Symposium, pp. 217-234, Aug. 2020.
〔18〕 Farkhani, R.M., and Jafari, S., Arshad, S., Robertson, W., Kirda, E., Okhravi, H. "On the effectiveness of type-based control flow integrity." In Proceedings of the 34th Annual Computer Security Applications Conference, pp. 28-39, Dec. 2018.

〈저 자 소 개〉



장 윤 수 (Yoonsoo Chang) 정회원
2021년 2월: 부산대학교 정보컴퓨터공학부 졸업
〈관심분야〉 컴퓨터공학, 정보보호

김 영 주 (Youngju Kim) 학생회원
2019년 8월: 울산과학기술원 컴퓨터공학 학사
2021년 3월~현재: 부산대학교 융합보안대학원 석사
〈관심분야〉 정보보안, TrustZone



권 동 현 (Donghyun Kwon) 정회원
2012년 2월: 서울대학교 전기컴퓨터공학과 학사 졸업
2019년 2월: 서울대학교 전기컴퓨터공학과 석박통합과정 졸업
2019년 3월~2020년 2월: 한국전자통신연구원 연구원
2020년 3월~현재: 부산대학교 정보컴퓨터공학부 조교수
〈관심분야〉 시스템 보안, 소프트웨어보안