CLC____________ Number_________

UDC____________ Available for reference ☐Yes ☐No

Southern University of Science and Technology

Undergraduate Thesis

Thesis Title:	**WASMYun: Control Flow Integrity for Ahead-of-Time WebAssembly**
Student Name:	**Zhaoqi Xiao**
Student ID:	**11810517**
Department:	**Computer Science and Engineering**
Program:	**Computer Science and Technology**
Thesis Advisor:	**Professor Yinqian Zhang**

Date: 2022.6.2

COMMITMENT OF HONESTY

1. I solemnly promise that the paper presented comes from my independent research work under my supervisor's supervision. All statistics and images are real and reliable.
2. Except for the annotated reference, the paper contents no other published work or achievement by person or group. All people making important contributions to the study of the paper have been indicated clearly in the paper.
3. I promise that I did not plagiarize other people's research achievement or forge related data in the process of designing topic and research content.
4. If there is violation of any intellectual property right, I will take legal responsibility myself.

Signature:

Date:

WASMYun: Control Flow Integrity for Ahead-of-Time WebAssembly

Zhaoqi Xiao

（Computer Science and Engineering　Thesis Advisor: Yinqian Zhang）

[**ABSTRACT**]: WebAssembly[1] is a novel binary format, which have been used as compilation target for many high-level languages. It supports three major execution methods-interpretation, Just-In-Time and Ahead-of-Time compilation. And we find Ahead-of-Time compilation may introduce many security problems like other binary formats.

In this work, we promoted a method to dynamically instrument Ahead-of-Time compiled module using Intel PIN[2], and this is the first work focusing on analyzing vulnerability in Ahead-of-Time compiled WebAssembly. By analyzing the information we get by instrumentation, we found some attack methods to compromise the WebAssembly virtual machine sandboxing mechanism by modifying Ahead-of-Time modules, in other words, creating malicious modules. To defend against these attacks, we implemented a prototype WASMYun-on top of the existing dynamically instrumenting approach, which can strength the existing control flow integrity of WebAssembly, and thus achieve a enhancement of the sandbox mechanism. Evaluation of our prototype shows that WASMYun can detect all the control flow anomalies, using branch target address validation for forward edge and shadow stack for backward edges. Its overall additional overhead is in the range of 3.63x to 11.96x. And the instrumented code causes the overhead ranged from 3.1x to 7.08x.

[**Key words**]: WebAssembly, Vulnerability Detect, Dynamic Instrumentation, Program Analysis

[摘要]：

WebAssembly 是一个新兴的二进制文件格式，并且已经被许多的高级语言支持作为编译的目标二进制文件。目前主要有三种执行 WebAssembly 的方式——解释执行、Just-In-Time 和 Ahead-of-Time 执行。其中 Ahead-of-Time 执行一直是二进制文件平台上最为大家所谨慎的，许多安全问题会在这种执行方式里面被引进。

在我们的工作当中，我们关注于 WebAssembly 的 Ahead-of-Time 执行方式。我们基于 Intel PIN 提出了一种专门针对这种运行方式的动态插桩方法，以来分析和防范其中可能出现的问题。我们提出了两种利用恶意的、经修改过的 Ahead-of-Time WebAssembly 模块修改 WebAssembly 虚拟机控制流，从而执行恶意代码的攻击。同时，我们也基于我们提出的动态插桩方法，实现了一个检测这类攻击的防御工具原型——WASMYun。WASMYun 针对控制流修改当中的前向边以及反向边都进行了防护，利用分支指令目标地址的检测针对前向边攻击，利用 shadow stack 进行反向边返回地址的防护。通过我们的评估，WASMYun 可以检测出我们提出的所有攻击，并且所有正常编译的 WebAssembly 都能在 WASNYun 下正常执行。WASMYun 跟直接通过 Ahead-of-Time 运行相比，会产生 3.63 倍到 11.96 倍的额外开销；如果仅仅关注我们插入的代码，换句话说，WASMYun 跟 Ahead-of-Time 加上 PIN 的开销相比，WASMYun 会产生额外的 3.1 倍到 7.08 倍的额外开销。

[关键词]： WebAssembly, 恶意软件检测, 动态插桩, 程序分析

Contents

1. Introduction

Dynamic instrumentation has been widely used in program analysis, to help detect possible anomalies and danger in the execution of the program. Before or after each instruction or sentence in a program, insert appropriate logic to save the state of the current program run and verify whether the current operation is legal in the current state at the right time. Although we can analyze the problems through a static way, it is often impossible for a static analyzing tool to obtain the information produced during the program's execution, which may lead to incomplete conclusions. In this case, dynamic analysis is very necessary.

Control flow integrity[3-8] and data flow analysis[9-11] are two representative usage scenarios for dynamic instrumentation. Control flow integrity ensures that a program does not run code that it should not have run, for instance, a piece of shell code which is injected by the attacker. During the execution of a program, PC register will jump between functions by call, jump and return instructions. We call the branch operations caused by call and jump the forward edge because it will be represented as a forward edge in control flow, and call these caused by return the backward edge for the same reason. Data flow analysis makes sure at the data level that, the program's data is always obtained from a trusted source and used in a valid sink (destination).

There have been several methods to achieve control flow integrity. For forward edge integrity, non-executable user stack[12] is one of them, which limits the control flow of current program cannot be changed to user stack, and it has been supported and enabled by default by most of the modern operating systems and compilers. Another method that works well for forward edge, mostly for call instruction, is function type checking[13], which requires collaboration between compile-time and runtime. During compile-time, the compiler analyzes the available function signature for branch target. A common used signature is the number and types of the parameters. And during runtime, extra instrumented code checks whether the branch target's signature matches one of the valid signatures. For backward edge integrity, shadow stack[5] is currently the more mature method, in which return address is kept by instrumented code. Return address validation will be performed when a return instruction is executed.

In this work, we introduce a method to do dynamic instrumentation for Ahead-of-Time compiled WebAssembly. Our design goal is to achieve control flow integrity by dynamic instrumentation. The key insight is that, by dynamic analyzing, discover some possible methods of attacking just for Ahead-of-Time compiled WebAssembly; further more, based on dynamic instrumentation, promote a method to detect and prevent these attacks. Specifically, by analyzing the code of WebAssembly-Micro-Runtime, a virtual machine for WebAssembly and our dynamic instrumentation, we have following observations. (1) WebAssembly-Micro-Runtime always loads Ahead-of-Time compiled WebAssembly into the first 2 Gigabytes of memory space (0x00000000 to 0x80000000). (2) WebAssembly-Micro-Runtime implements function signature checking and indirect jump for forward edge integrity. (3) WebAssembly-Micro-Runtime separates execution stack and data stack for WebAssembly. What we call execution stack stores the return address of the Ahead-of-Time compiled WebAssembly and data stack is in linear memory, which we will cover in detail in security evaluation section. Because of this separation, it becomes difficult for attackers to overwrite return address, and thus impossible to perform Return-Oriented Programming (RoP) attacks[14]. (4) WebAssembly-Micro-Runtime's sandboxing mechanism for Ahead-of-Time execution is so weak and almost non-existent that developers need to specifically state that running an untrusted Ahead-of-Time module is totally unsafe. (5) There have been many Ahead-of-Time compiled WebAssembly usage scenarios in commercial use, e.g.SSVM[1]. And we perform several attacks successfully by modifying Ahead-of-Time compiled WebAssembly module and run it on WebAssembly-Micro-Runtime, which causes invalid control flow change. Because there is still no method to protect the control flow integrity especially for Ahead-of-Time WebAssembly, we design this approach to enhance sandbox strategy and control flow integrity for Ahead-of-Time WebAssembly.

Compared to other methods, which may perform static analysis to Ahead-of-Time WebAssembly[15], because our approach gets the runtime information, more kinds of protection can be achieved. First, to achieve forward edge integrity, we limit the branch target during runtime. However, these information keeps unknown until load time, which causes static analyzing unable to perform this analysis. Second, static analysis cannot always get the accurate return address, while shadow stack can, which leads to more accurate backward edge integrity in our approach.

To validate our idea, we implement WASMYun (WebAssembly Control Flow Integrity by D**Y**namic Instr**U**mentatio**N**), a prototype based on Intel PIN[2]. It can perform dynamic instrumentation on Ahead-of-Time compiled module on WebAssembly-Micro-Runtime. Moreover, it implements forward edge integrity by checking branch address and shadow stack

[1]https://www.secondstate.io/ssvm/

to validate backward edge integrity. To evaluate it's effectiveness, we design two attack methods for forward edge and backward edge correspondingly and implement two malicious Ahead-of-Time compiled WebAssembly modules. We proved that, our tool can successfully detect and prevent execution of these malicious code. To evaluate its efficiency, we test it on PARSEC benchmark[16]. The result shows that, compared to execution without PIN, WASMYun causes extra 3.63x to 11.96x additional overhead; compared to execution with PIN while without instrumentation, the overhead is in the range of 3.1x to 7.08x.

To summarize, we make following contributions:

- We design a novel approach to perform dynamic instrumentation for Ahead-of-Time compiled WebAssembly.
- We promote two feasible attack methods related to control flow integrity, including forward edge attack and backward edge attack, for Ahead-of-Time compiled WebAssembly.
- We implement a prototype WASMYun based on Intel PIN.
- We implement two modified Ahead-of-Time compiled modules to attack against forward edge and backward edge separately.
- We evaluate our prototype with PARSEC benchmark and the modified Ahead-of-Time compiled WebAssembly modules.

2. Background

In this section, we first provide some background knowledge about WebAssembly, including its basic design and outstanding features. Then, we will discuss the existing security problem in WebAssembly and its corresponding countermeasures.

WebAssembly Overview. WebAssembly is a binary instruction format for a stack-based virtual machine. It is designed for portable compilation platform and available on both web browser and isolated virtual machine. It has the advantages of being fast, safe and compact.[17] Even though it is announced in 2015[2], it has been implemented by all primary browsers in 2017[18]. WebAssembly is designed for compilation target; and there have been many compilers supporting compiling different languages to WebAssembly like rustc

[2]https://github.com/WebAssembly/design/issues/150

for Rust, and Emscripten and wasi-sdk for C/C++. Besides compilers, there are some isolated virtual machines independent from web browsers like Wasmtime and WebAssembly-Micro-Runtime. With specific system interfaces, these isolated virtual machines can run WebAssembly Module outside web browsers.

Nowadays, many organizations try to use WebAssembly to isolate different tenants. Because the execution of different WebAssembly code in a single WebAssembly virtual machine is independent and is isolated by the virtual machine, any behaviour which breaks the isolation of virtual machine will cause a malicious client to escape from the sandbox environment created by virtual machine, and then allow the client to control the server by change the control flow of the WebAssembly module.

When a piece of source code want to be run on a WebAssembly virtual machine, it should first be compiled from high-level source code to WebAssembly bytecode. During this phase, compilers like rustc and LLVM will compile the high-level language source code and link them with proper system interface library like Emscripten and WebAssembly System Interface to produce WebAssembly module. A WebAssembly Module is a binary format file, storing the instructions run on WebAssembly virtual machine in binary form.

A WebAssembly virtual machine run a WebAssembly module by several methods: a. Interpret b. Just-In-Time compilation c. Ahead-of-Time compilation. By interpreting, the virtual machine uses a set of data structures to simulate the execution process of a WebAssembly module; by Just-In-Time compilation, the virtual machine first parses a WebAssembly module, then generates intermediate representation and uses a compiler back-end to transform intermediate representation to native code running on physical CPU; by Ahead-of-Time compilation, instead of obtaining a WebAssembly module from outside the virtual machine gets a set of Ahead-of-Time compiled native instructions, and execute it directly. In WebAssembly-Micro-Runtime, the core virtual machine-iwasm is in charge of the compilation process.

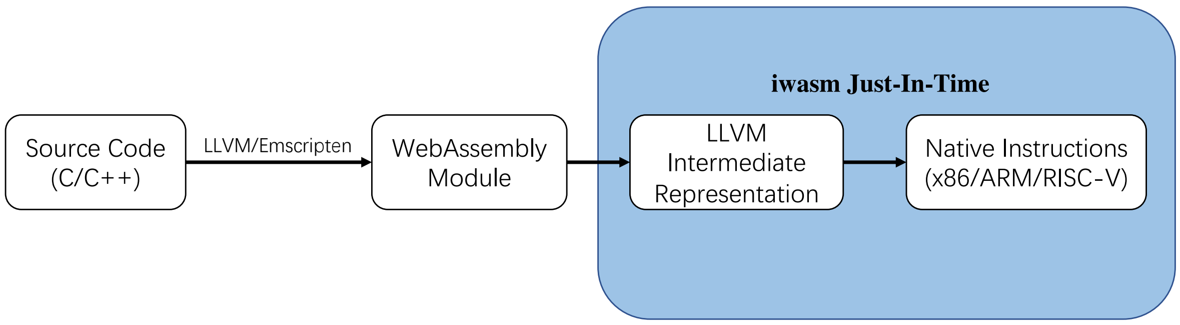
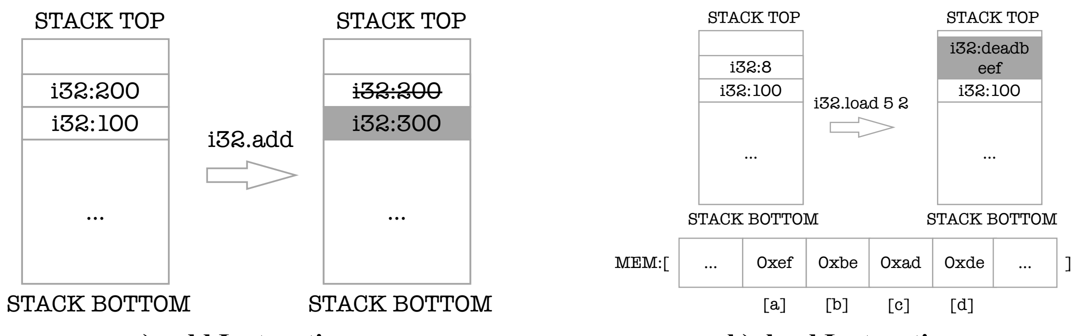


Figure. 1 WebAssembly Just-In-Time Execution Process

For high-level languages like C/C++, the compilers do not check the memory bounds when they compile source code to native code, making vulnerabilities like buffer overflow[4, 19]. These vulnerabilities can change the control flow of normal program to perform anomalies. Nevertheless, the common anomaly like changing return address to modify the control flow does not work perfectly on WebAssembly because of the different memory layout between WebAssembly virtual machine and register-memory based machine.

Stack-Based Virtual Machine. Most register-memory based machines use register and memory address as the operands of the instructions, like add $t0, $t1, $t2, which means do addition for $t1 and $t2, and store the result into $t0. The WebAssembly virtual machines use another stack-based instructions. To be specific, all the operands of these instructions are stack-based. It is like a common algorithm design problem, calculating arithmetic expressions. The instruction 'i32.add' means popping two 32-bits integers from the stack, adding them and storing the result back to the stack. Also, because WebAssembly uses linear buffer as memory, there are instructions related to memory in WebAssembly like load and store. The instruction 'i32.load [align] [offset]' means popping one operand from the stack, adding this operand to offset to get the final offset of memory load and pushing the data in memory into the stack; the parameter 'align' here is only for prompting purpose. In picture.2b, the operand in the top of the stack is 8, and the offset is 2. The final offset calculated is 10 with hexadecimal form of 0xa. So the data in the offset of 0xa in the memory will be loaded and popped into the stack. Meanwhile, as WebAssembly virtual machine is a little-endian machine, data 0xdeadbeef will be loaded into the stack.



a) add Instruction **b) load Instruction**

Figure. 2 Stack-Based Virtual Machine

Linear Memory Buffer. As mentioned above, WebAssembly is initially designed for web browsers, among which the most popular language is JavaScript. Besides the interfaces for JavaScript calling the functions in a WebAssembly module, there should be an another data structure to pass the information between JavaScript virtual machine and WebAssembly virtual machine. Absolutely, the parameter is not enough; so another choice is a memory buffer[3]. This memory buffer is just used as an array, and can be accessed by an index.

Local and Global Variables. In addition to stack and linear memory, there are some extra local and global variables storing the data used in WebAssembly. These data can be accessed by instructions related to ’global’ and ’local’. For instance, ’local.get [index]’ will get the value of local variable with index, and push it into the stack.

Typed Variables. In most native languages, the operands of instructions are not typed, which means the instructions view them as raw data. Nonetheless, the operands in WebAssembly is typed, no matter for the data in the stack, linear memory, local and global variables. The work by Lehmann *et al.*[20], summarizes how these typed variables maintain the availability of static checking before binary execution; as well as how the primitive four types-32 bits, 64 bits integer and float-in WebAssembly are lowered to from the source-level complex types during the compilation time to achieve the simplicity and security features in the WebAssembly.

Indirect Calls. The WebAssembly keeps all the functions’ information in the so-called table section[20]. Unlike the most direct jump instructions from one procedure to another, WebAssembly uses indirect calls3 between procedures. When instruction ’call_indirect’ meets, the virtual machine needs popping one number from the stack and use it as the index to search in Table section, determining which function to be called. Besides finding and jumping to the correct code, the call_indirect instruction also passes two extra arguments for type checking of functions to ensure that the correct function is called during the control flow.

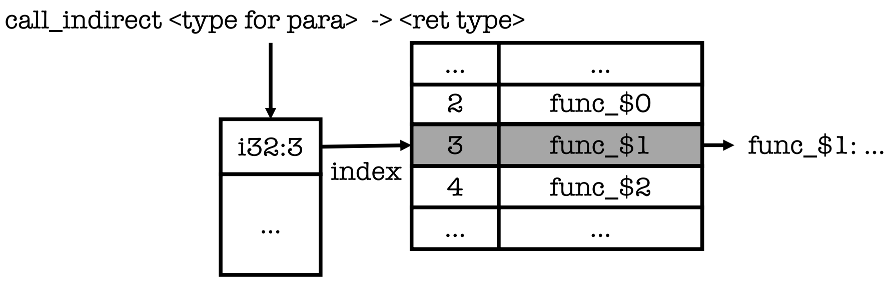


Figure. 3 Indirect Calls

WebAssembly System Interface (WASI). Before the independent virtual machines appear, the virtual machines are mostly embedded in browsers and serve for JavaScript virtual machine. Interaction with system calls needs to be done through JavaScript virtual machine. The WebAssembly System Interface provides a set of standard for isolated virtual machine on how they interact with host operating system. More specifically, which system calls vir-

[3]https://webassembly.github.io/spec/core/syntax/modules.html#memories

tual machine can provide to WebAssembly modules and how they be mapped to system calls or standard library in host operating system.

Emscripten is one library providing system interfaces and libc library functions for compiling high-level languages to WebAssembly. Later on, WebAssembly System Interface is promoted[4].

Different virtual machines support different kinds of library functions and standards. In our work, we will focus on WebAssembly System Interface.

WebAssembly Security. The WebAssembly has achieved several simple control flow integrity. (1) Indirect function call we introduced before ensures the target function call is valid. (2) Function signature checking inspects whether the signature of target function matches the signature recorded during compilation. The compiler first calculates and records the signature sig_f of function f(), instruments code to verify function signature. When it comes to runtime, main function will inspect whether the signature of function f_runtime() matches sig_f.

Besides existing safety strategy, there are still some vulnerabilities in WebAssembly. In the work of Lehmann *et al.*[20], they analyzed the differences between memory layout of WebAssembly and traditional x86 PC. For WebAssembly, two parts of data are divided, which are called *managed data* and *unmanaged data*. The managed data, including local and global variables, values inside the stack manipulated by instructions, and other implicit data handled directly by virtual machine like return addresses, can only be interacted through instructions. Like by instruction 'i32.add', we can pop two 32 bits integers from the stack and store one value, the result of the addition, into the stack. However, during the overall process, we can never get an exact address of these data so that we could modify or read them from memories. Specifically, all the data are managed by the instructions in WebAssembly module directly, and no matter what modifications are made in the execution states, these data cannot be controlled by the environment outside the module itself, except the module wants so.

Apart from the managed data, there is still some unmanaged data referring specifically to the data stored in linear memory. These data can be influenced by the vulnerable behavior of WebAssembly modules. Like traditional compiling process for x86 machines, when compiling the high-level source code to WebAssembly, the compilers will create memory area such as stack, heap and static area for call stack, dynamic life-time data and static data.

[4]https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/

For coherence, we also use the names in *Everything Old is New Again*[20]-managed stack and unmanaged to specify the stack which is manipulated by instructions and the stack created by compilers in the linear memory.

For the data in unmanaged stack, it can cause anomalies by traditional vulnerabilities occurred in C/C++ programs like buffer overflow and string format vulnerability. In this work, buffer overflow is used to modifying the existing data like file descriptors and function pointers to change the I/O stream and control flow of the program. All these anomalies performed are not detected or terminated by the major three virtual machines, Wasmtime, Wasmer and WebAssembly-Micro-Runtime. Programs with these vulnerabilities are all compiled from C/C++ source code by wasi-sdk-14 with clang and LLVM. It is implied that, even though there are the design and claim of security in WebAssembly, some vulnerabilities in C/C++ program still occurred in WebAssembly. Even till nowadays, hardly can virtual machines ever detect and terminate these anomalies.

3. Our Approach

3.1 Insight

Our design goals are finding some vulnerabilities especially for Ahead-of-Time compiled WebAssembly and promoting corresponding possible solution to them. In order to achieve this goal, we think of dynamic analysis, which is often used in program analysis. So the first and most fundamental problem that needs to be solved is how to instrument into Ahead-of-Time compiled WebAssembly. Our solution is to use Intel PIN because (1) it is currently a more mature instrumentation platform for x86-64 architecture and (2) it supports self-modifying code instrumentation, which is the presence form of Ahead-of-Time compiled WebAssembly in WebAssembly-Micro-Runtime.

By analyzing the native code Ahead-of-Time compiled from WebAssembly, as we expected, there are still some vulnerabilities in Ahead-of-Time compiled WebAssembly that could be exploited by attackers. An attacker could change the control flow of WebAssembly virtual machine by having it run a modified Ahead-of-Time module, either for the forward edge or the backward edge.

To prevent attackers from realizing these attacks, once again, we perform additional checks on the execution state of the program using dynamic instrumentation. They are executing text only for forward edge and shadow stack for backward edge separately. Finally, we succeed in avoiding the attack we successfully executed on WebAssembly virtual machine.

3.2 Overview

Picture.4 shows the design of WASMYun. WASMYun is divided into three major parts: (1) the analyze instrumentation first analyzes the current virtual machine state in two aspects, code segment and WASI-Related functions. Code segment is the exact memory space which is allocated by WebAssembly-Micro-Runtime to load Ahead-of-Time compiled WebAssembly; and WASI-Related functions are valid targets besides code segment. (2) Checking instrumentation part obtains this required information from analysis instrumentation parts and instruments checking code into Ahead-of-Time compiled WebAssembly. (3) Execution engine then executes Ahead-of-Time compiled WebAssembly along with the instrumented code.

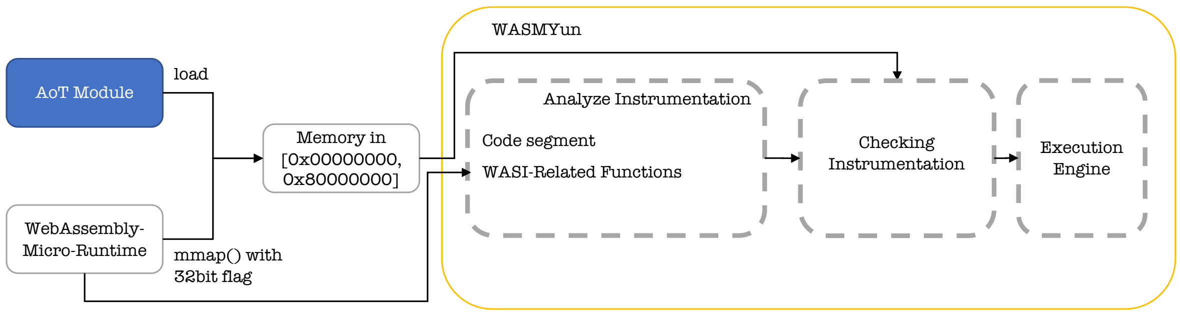


Figure. 4 Overview of WASMYun

3.3 Analyze Instrumentation

Analyze instrumentation is responsible for extracting the necessary information from the current virtual machine. The necessary information, in another word, the valid branch targets for Ahead-of-Time compiled WebAssembly can be extracted from the WebAssembly virtual machine by dynamic instrumentation. Code segment information consists of code segment offset and code segment length. WASI-Related functions are the function entrance for WASI system call.

3.4 Checking Instrumentation

With information collected by analyze engine, instrumentation engine instruments branch instructions in Ahead-of-Time compiled WebAssembly. They are predominantly three types of branch instructions: (1) JUMP, (2) CALL, and (3) RETURN. In addition to invalid branch checking, another forbidden instruction is syscall, which should occur in an Ahead-of-Time compiled WebAssembly. As Intel PIN focuses on x86-64 architecture computer, WASMYun also focuses only on x86-64 platform. Among these types of branch instruction, JUMP and CALL are for forward edge branch and RETURN for backward edge. The instrumentation for JUMP inspects the validation of target address. The one for RETURN will pop one return address (RA) from shadow stack and match it with the actual return address. CALL instruction is the most complex one, because the instrumentation for it should complete both branch target checking and maintaining shadow stack. When a call instruction is performed, the desired (valid) return address need to be pushed into shadow stack.

3.5 Execution Engine

After the instrumentation, WASMYun executes the WebAssembly virtual machine with instrumented code. This part is all done by the Intel PIN and we put it here just for displaying the complete design of our approach.

4. WASMYun

In this section, we present the design details of WASMYun.

4.1 Locate Ahead-of-Time WebAssembly

Before we instrument, we weed to know the location of Ahead-of-Time compiled WebAssembly in the memory because it does not belong to the text segment of WebAssembly virtual machine.

When PIN instruments a program, it will first analyze the whole program to obtain execution information like the location of text segment. However, an Ahead-of-Time compiled WebAssembly is not in text segment of virtual machine, which means the PIN cannot instrument this part directly. Instrumenting into Ahead-of-Time WebAssembly is difficult as well because its format is not standard Extensible Linking Format (ELF).

The solution we choose is instrumenting code after load time of WebAssembly virtual machine. After load time, the exact Ahead-of-Time compiled WebAssembly is loaded into memory and can be treated as a self-modifying code.

The Intel PIN is able to instrument for self-modifying code. However, we need to know the exact address of this self-modifying code in the memory. WebAssembly-Micro-Runtime, the virtual machine we focus on in our approach, loads the Ahead-of-Time compiled WebAssembly into a block of memory, which is allocated by mmap() with 32BIT flag. The 32BIT flag for mmap() requires mmap() mapping a block of memory in the first 2 Gigabytes in memory space. As a result, Ahead-of-Time compiled WebAssembly will be put in the address of [0x00000000, 0x80000000].

WASMYun uses PIN_SetSmcSupport() function to enable the support for self-modifying code and it only instruments the code in the memory of [0x00000000, 0x80000000].

4.2 Image Level Instrumentation

The analyze instrumentation is for obtaining necessary information which is produced only in execution time. It consists of mainly two parts: (1) code segment information and (2) WASI-related function information.

Code segment information is the accurate memory address range for for Ahead-of-Time compiled WebAssembly. We know that an Ahead-of-Time compiled WebAssembly will be put in the memory of [0x00000000, 0x80000000]. However, we do not know the exact address of the Ahead-of-Time compiled WebAssembly only after the load time. That is why we need dynamic instrumentation to obtain this information. And this information is used to limit the branch target address.

WASI-related function information is some function entry pointer in WebAssembly virtual machine. For WebAssembly-Micro-Runtime, all the WASI call will be redirected to the function aot_invoke_native() in WebAssembly-Micro-Runtime. So the information here is the pointer of aot_invoke_native() in WebAssembly-Micro-Runtime.

To get the information, we instruments code in image level. The image level in Intel PIN is defined as single ELF file, which can be an executing file, dynamic link library or static link library. In image level, we can search for specific function and monitoring the information of arguments or return value of the function.

To get the information of code segment, we instruments for load_text_section() function. One of this function's argument is AOTModule, which contains code segment information of an Ahead-of-Time compiled module. We need to do instrumentation for both entrance and exit of this function because the argument information can be only got in entry of a function and the code segment can be only determined after completion of this function. The instrumented code before this function will get the pointer pointed to the AOTModule and the one after this function will get the exact code segment information. The segment information contains the start address and the length for the code segment. For WebAssembly-Micro-Runtime we used, the offset of the start address is 0xf0 and the offset of code segment length is 0xf8.

4.3 Instruction Level Instrumentation

The checking instrumentation is to check the legitimacy of the target address of branch instructions and maintain shadow stack. It do instrumentation for JUMP, CALL and RETURN instructions. In order to avoid Ahead-of-Time compiled WebAssembly executes system call directly, WASMYun also instruments syscall instructions.

JUMP. For single JUMP instruction, the instrumented code checks whether the branch target address is in code segment, whose information we have obtained from image level instrumentation. If the branch target address is not in the code segment, the instrumented code will additionally check if the branch target address is a WASI-related function. If both checking fails, this JUMP instruction would be considered an illegal jump. WASMYun will terminate its execution and record related information about this instruction.

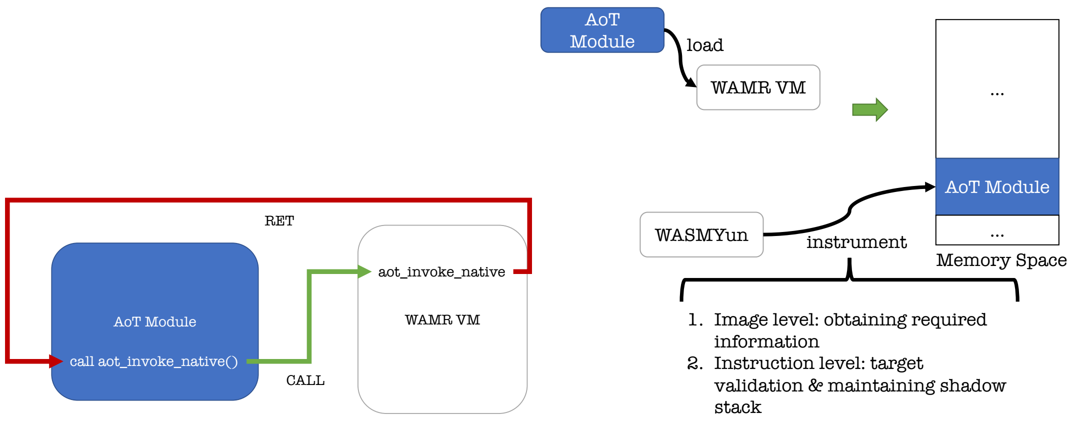
CALL. Compared to JUMP, the instrumentation for CALL instructions is more complex. The difference between CALL and JUMP instruction in x86-64 is that, CALL instruction is for function calling, which means it will maintain the stack frame including pushing return address and saving caller-saved registers.

In addition to ensuring the legitimacy of the branch target address which is the same as JUMP, shadow stack also needs be maintained at the same time. The shadow stack is a stack allocated in WASMYun storing the pointer-typed data. For each call instruction, WASMYun uses INS_NextAddress() in Intel PIN to record the address of the next instruction and pushes it the shadow stack.

RETURN. The RETURN address will return from the current function, recovering the stack to the state of last function. WASMYun instruments the code which pops one address from the shadow stack and matches it with the actual return address. If they are not same, which means the return address has been modified, WASMYun will terminates the execution and record related information.

Nevertheless, there is a special case that will cause the CALL instruction that we instrumented inside Ahead-of-Time compiled WebAssembly to not match the RETURN instruction. In Picture.5a, when a CALL instruction is directed to a WASI-related function, the RETURN instruction is in WebAssembly virtual machine instead of Ahead-of-Time compiled module, which we do no instrumentation. To solve this problem, for each CALL instruction whose branch target is a WASI-related function, we will pop the shadow stack in the CALL instruction instead of RETURN instruction.

Besides, we need to instrument another single instruction in WebAssembly-Micro-Runtime. Because the control of CPU should be passed to Ahead-of-Time compiled WebAssembly by a CALL instruction in virtual machine, we need to verify whether the return address of the RETURN instruction in top function of Ahead-of-Time compiled WebAssembly matches the next instruction address of this CALL instruction.



a) Not Matched Call and Return **b) WASMYun Implementation**

Figure. 5 Not Matched Stack Manipulations and Implementation of WASMYun

Syscall. Another possible attack which can redirect the control flow of the virtual machine is using syscall instruction. Theoretically, Ahead-of-Time compiled WebAssembly should not contain syscall instruction because all the system call should be invoked through WebAssembly System Interface. So we disable all the syscall instruction. Whenever WASMYun detects syscall instruction, the execution of program will be terminated and related information will be recorded.

5. Implementation

In this section, we introduce some implementation details of WASMYun.

WASMYun is totally based on Intel PIN and written in C++17 standard. The main() function consists by 4 parts: (1) image level instrumentation function Image(), (2) instruction level instrumentation function Instruction(), (3) PIN_SetSmcSupport() which enable self-modifying code support of PIN and (4) PIN_StartProgram() the execution engine provided by PIN.

In Image(), WASMYun first finds two key functions-load_text_section() and aot_invoke_native(). The previous function is for getting the code segment information and the latter one is for getting the WASI-related functions' information. Function finding is done by RTN_FindByName(), which can find routine in specific image by name.

In Instruction(), WASMYun only focuses on the instruction with addresses in range [0x00000000, 0x80000000]. INS_IsControlFlow() judges whether a instruction is a branch. For all the branch instruction except RETURN, WASMYun instruments the code responsible for branch target validation. Before each CALL instruction, WASMYun pushes the address of the next instruction into shadow stack; and before each RETURN address, it pops one address from shadow stack and check whether the actual return address got by macro IARG_BRANCH_TARGET_ADDR is the same as the one got from shadow stack. When a syscall instruction is found by WASMYun, WASMYun terminates this process immediately.

6. Evaluation

In this section, we evaluate our prototype WASMYun, aiming to answer the following questions: (1) can it detect possible attacks for Ahead-of-Time compiled WebAssembly and (2) what is its performance?

Experiment Setup. All evaluation was done on work station with 8-core, 16 thread Intel i7-10700 processors. The work station has 32G memory. The operating system is Ubuntu 20.04.4 with kernel 5.4.0. The WebAssembly-Micro-Runtime and WAMR-Compiler used is tagged by {f8ee05db}.

Security Evaluation. Before evaluating WASMYun, we would like to introduce the attacking method especially to Ahead-of-Time compiled WebAssembly.

To modify the control flow of a program, the common ways are changing either forward edge or backward edge. For forward edge, an attacker can insert invalid CALL, JUMP or Syscall instruction into Ahead-of-Time compiled WebAssembly, which can lead the control flow modification of the virtual machine.

Listing.1 shows a piece of code which is written in C++. A long long typed integer is stored into memory. After we compile it to WebAssembly using wasi-sdk-14 (LLVM), and further compile the WebAssembly to Ahead-of-Time compilation module, we can get a module with native x86-64 instructions in listing.2. In the 6th line of native code, a number 0x5555555f18f7 is moved in register $rdx.

```cpp
int main() {
        long long a = 0x5555555f18f7;
        return 0;
}
```

Listing 1: C++ Code

```
1  mov     0x10(%rdi),%rax
2  mov     0x158(%rax),%rcx
3  mov     0x1a8(%rax),%eax
4  add     $0xfffffff0,%eax
5  movl    $0x0,0xc(%rcx,%rax,1)
6  movabs  $0x5555555f18f7,%rdx
7  mov     %rdx,(%rcx,%rax,1)
8  xor     %eax,%eax
9  retq
10 nopl    (%rax)
```

Listing 2: Native Code

```
1  mov     0x10(%rdi),%rax
2  mov     0x158(%rax),%rcx
3  mov     0x1a8(%rax),%eax
4  add     $0xfffffff0,%eax
5  movl    $0x0,0xc(%rcx,%rax,1)
6  movabs  $0x5555555f18f7,%rdx
7  mov     %rdx,(%rcx,%rax,1)
8  callq   *(%rdx)
9  retq
10 nopl    (%rax)
```

Listing 3: Native Code Modified

If we modify the instruction in line 8, from xor %eax %eax to callq *(%rdx), the program will redirect the control flow of the program to .text segment of virtual machine. Likely, we can also change the control flow of the program by introducing a malicious JUMP or Syscall.

Another way, the backward edge can be modified by inserting malicious POP and PUSH instructions. During execution of Ahead-of-Time compiled WebAssembly, even though all the data are stored in linear memory, the return address is kept in the stack of virtual machine. Ahead-of-Time compiled WebAssembly uses CALL, PUSH and POP instructions to manipulate this stack. Therefore, we can insert extra POP and PUSH instructions in Ahead-of-Time compiled WebAssembly to overwrite the original return address. In Picture.6, the most right instructions are separated by two colors, black and red. The black part is the normal instructions of Ahead-of-Time compiled WebAssembly and the red ones are inserted by the attacker. As the execution stack of an Ahead-of-Time compiled WebAssembly stores no data, there is only return address in this stack. With a pair of simple POP and PUSH instruction along with a bad return address in register rax the attacker can alter the return address to change the backward control flow.

Our evaluation shows that, all these attacks, no matter through forward edge or backward edge, can be detected by WASMYun. And all the normal Ahead-of-Time compiled WebAssembly can be run successfully.

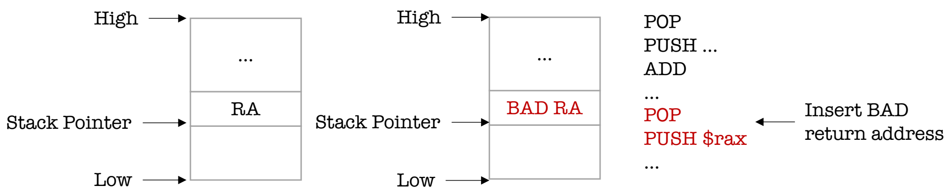


Figure. 6 Backward Edge Exploit

Performance Evaluation. We evaluated WASMYun on PARSEC Benchmark[16]. The result shows that the additional overhead varies with different programs. The Blackscholes executes 924032861068 instructions under native input, and 51611088369 of them are branch instructions, which occupy 5.59%. The Hashes, in which WASMYun causes the most overhead, has 9.81% branch instructions. Because WASMYun only cares about branch instructions, programs with more branch instructions will perform more instrumentation, which causes more overhead. We can find the whole system causes overhead ranging from 3.63x to 11.96x.

Table. 1 PARSEC Benchmark Result

program	AoT	AoT + WASMYun	(AoT + WASMYun) / AoT
Blackscholes	133.58s	485.45s	3.63x
Fluidanimate	726.86s	4097.98s	5.64x
Freqmine	414.64s	2211.12s	5.33x
Swaptions	273.01s	1032.48s	3.78x
Bodytrack	425.15s	3565.86s	8.38x
Hashes	23.59s	282.06s	11.96x

In addition to overhead of WASMYun compared with execution under Ahead-of-Time mode directly, we calculate the one compared with execution with PIN as well. In picture.7, the blue bar is the overhead of WASMYun compared with PIN. Because WASMYun is based on PIN, besides the overhead caused by the whole system, we also want to learn the overhead caused by our instrumented code. The result shows that our instrumented code causes the overhead ranging from 3.1x to 7.08x. The result matches the branch instructions count. In Blackscholes, which has 5.59% branch instructions, the overhead is 3.1x; in Hashes with 9.81% branch instructions has the overhead of 7.08x.

Another important overhead comes from self-modifying code analyzing. Because Ahead-of-Time compiled WebAssembly is treated as a piece of self-modifying code in WebAssembly virtual machine, during the execution of WebAssembly virtual machine, WASMYun should check the possible occurrence of self-modifying code. That is why the overhead of WASMYun is higher than some of tools[5] also based on PIN.

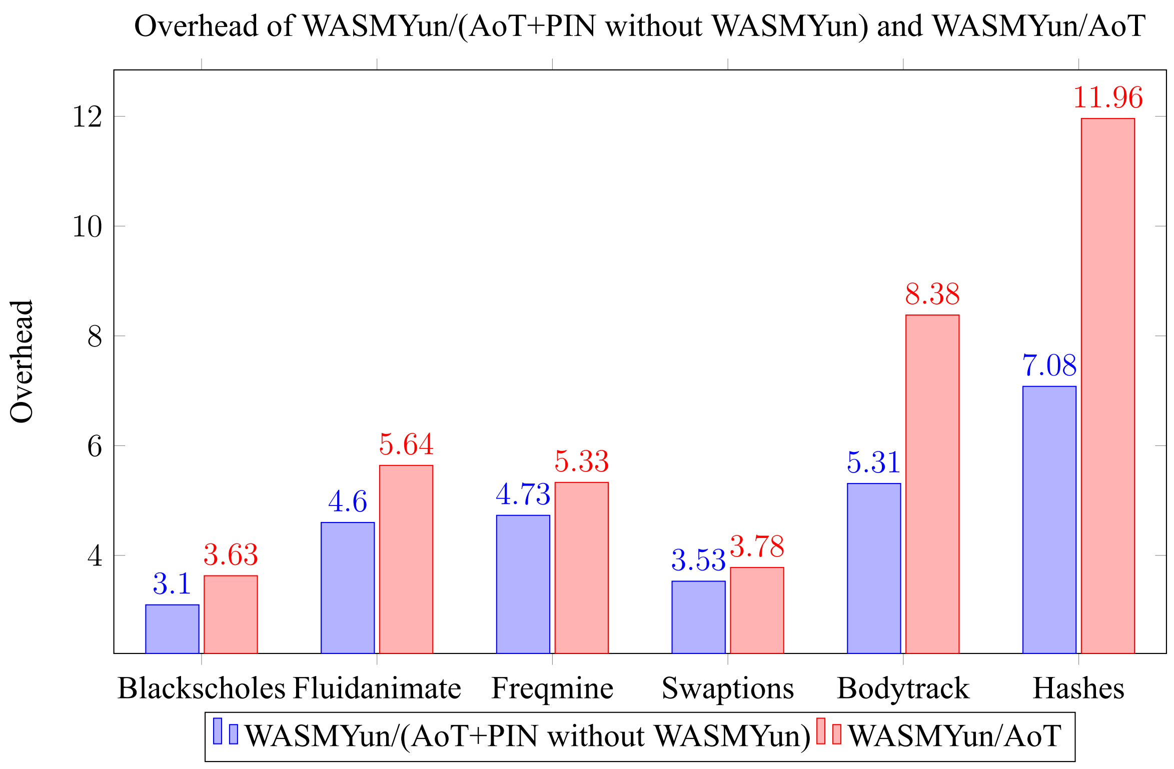


Figure. 7 Overhead Comparison

Summary. Based on these two experiments, the answer to the first question is yes: our prototype can detect the possible attacks which utilize malicious Ahead-of-Time compiled WebAssembly. And the answer to the second question is that, our prototype produces additional overhead ranging from 3.63x to 11.96x compared to Ahead-of-Time execution directly; the overhead of our instrumentation ranges from 3.1x to 7.08x. This overhead is acceptable among the tools using dynamic instrumentation.

7. Discussion

In this section, we discuss the limitations of our current prototype. In addition, we will give some suggestions to the potential future works.

Support for More WebAssembly Virtual Machine. The current prototype of WASMYun can only support the WebAssembly-Micro-Runtime. Theoretically, it could support more platform like WasmTime and Wasmer. However, to support each of the virtual machine, we need to analyze the code of these virtual machines manually. Most of the virtual machines of WebAssembly are developed by Rust, which is more difficult for us to analyze than C/C++.

Support for Optimized Virtual Machine. In our current version of prototype, WAS-

MYun can only support WebAssembly-Micro-Runtime without optimization. During the compilation optimization, many functions will be emitted, in which aot_invoke_native() is included. Without the pointer to aot_invoke_native(), we cannot generate white-list for valid branch targets in virtual machine. However, these rules can be specified by the user if the user can decide where these key functions' pointers are.

Data Access Control. For a complete sandbox, besides control flow access limitation, another important limitation is data access. In our experiment, a valid Ahead-of-Time compiled module needs to access memory of virtual machine, including stack and heap. For stack access, the Ahead-of-Time compiled module needs to manipulate its runtime stack, in which there are return addresses. And there are also many other push and pop instructions which will interact with the stack memory. In addition to stack access, the Ahead-of-Time compiled module will also access the heap memory in the virtual machine. With more observation of execution of runtime, we can produce a list of valid memory access addresses for WASMYun to achieve data access control.

8. Related Work

Vulnerabilities of C/C++ Programs in WebAssembly. The work by Lehmann et al[20] summarized how some vulnerabilities produced by C/C++ programming language perform on WebAssembly, including how buffer overflow in C/C++ programs can influence the running process of WebAssembly. However, as some protection strategies achieved by WebAssembly, these attacks has few scenarios in the real world. For instance, even though we can modify a function pointer stored in stack by buffer overflow, we can hardly ever change the control flow of the program to perform vulnerable actions by overwriting the correct function pointer because of the dynamic function type-checking. Other strategies like separation of running stack and data stack also prevent the attacker to modify the control flow to whatever they want.

WebAssembly Dynamic Instrumentation Framework. Wasabi[21] is a dynamic instrumentation framework proposed by Daniel as well. It focuses on the instrumentation WebAssembly on bytecode level. By dynamically instrumenting instructions into WebAssembly module, specific actions such as monitoring data access and branches can be achieved. Because it performs the instrumentation on WebAssembly level, when the instrumented modules run on the physical machine, these instrumented code will produce higher overhead compared to PIN/Valgrind such native code instrumentation tools. Its evaluation with compute-intensive benchmark programs and real-world web applications shows 1.02x to 163x runtime overhead. Furthermore, for the malware promoted in our evaluation section, Wasabi does not handle these attacks well because of the different threat model.

SFI safety for native-compiled WebAssembly VeriWasm[15] is a static analysis tool for native-compiled WebAssembly module. By analyzing the linear memory, stack layout, global variables and control flow, VeriWasm makes sure that, a native-compiled WebAssembly module is safe and satisfies the safety strategy of WebAssembly. However, such method of static analyzing has some shortcomings. First of all, without runtime information like branch targets and return addresses, the measures to protect control flow integrity are simple and many malicious behavior cannot be detected successfully by VeriWasm. False-positive judgement is another problem. VeriWasm is clarifies to be sound, which means the WebAssembly modules labeled malicious by VeriWasm is truly malicious, while some valid and safe WebAssembly module can be probably labeled malicious by VeriWasm as well. Further more, a WebAssembly module contains malicious code does not guarantee that this piece of malicious code will be truly executed and this is difficult for VeriWasm to recognize.

RoP Attack Detection. ROPDefender[5] was proposed by Lucas to defend the Return-Oriented Programming (RoP) attack by dynamic instrumentation. ROPDefender is based on Intel PIN as well. It maintains a shadow stack and monitors all the call instructions and return instructions when process executing. Every time a direct call or return is performed, ROPDefender will push or pop the shadow stack which stores the return address, verifying whether the practical return address is the same as the expected one. ROPDefender is successful in detecting RoP attack, but with few ability to handle this attack in Ahead-of-Time compiled WebAssembly module because the module is not treated as a normal program in text segment. In addition to the support for instrumenting Ahead-of-Time compiled WebAssembly, we also add forward edge checking besides protecting backward edge by shadow stack.

9. Conclusion

In this paper, we present a novel way to perform dynamic instrumentation to Ahead-of-Time compiled WebAssembly based on Intel PIN. This is the first work focusing on such instrumentation targets. We have also promoted two possible attacks occurring in modern WebAssembly virtual machine which utilize malicious Ahead-of-Time WebAssmebly. Our evaluation results show that WASMYun can detect these attacks successfully and the overhead of WASMYun is acceptable among dynamic instrumentation based programming analyzing tools.

References

[1] HAAS A, ROSSBERG A, SCHUFF D L, et al. Bringing the Web up to Speed with WebAssembly[J/OL]. SIGPLAN Not., 2017, 52(6): 185-200. https://doi.org/10.1145/3140587.3062363. DOI: 10.1145/3140587.3062363.

[2] LUK C K, COHN R, MUTH R, et al. Pin: building customized program analysis tools with dynamic instrumentation[J]. Acm sigplan notices, 2005, 40(6): 190-200.

[3] BUROW N, CARR S A, NASH J, et al. Control-Flow Integrity: Precision, Security, and Performance[J/OL]. ACM Comput. Surv., 2017, 50(1). https://doi.org/10.1145/3054924. DOI: 10.1145/3054924.

[4] COWAN C, PU C, MAIER D, et al. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks.[C]//USENIX security symposium: vol. 98. [S.l. : s.n.], 1998: 63-78.

[5] DAVI L, SADEGHI A R, WINANDY M. ROPdefender: A Detection Tool to Defend against Return-Oriented Programming Attacks[C/OL]//ASIACCS '11: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. Hong Kong, China: Association for Computing Machinery, 2011: 40-51. https://doi.org/10.1145/1966913.1966920. DOI: 10.1145/1966913.1966920.

[6] CARLINI N, BARRESI A, PAYER M, et al. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity[C/OL]//24th USENIX Security Symposium (USENIX Security 15). Washington, D.C.: USENIX Association, 2015: 161-176. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini.

[7] ABADI M, BUDIU M, ERLINGSSON Ú, et al. Control-Flow Integrity Principles, Implementations, and Applications[J/OL]. ACM Trans. Inf. Syst. Secur., 2009, 13(1). https://doi.org/10.1145/1609956.1609960. DOI: 10.1145/1609956.1609960.

[8] ABADI M, BUDIU M, ERLINGSSON Ú, et al. Control-Flow Integrity[C/OL]// CCS '05: Proceedings of the 12th ACM Conference on Computer and Communications Security. Alexandria, VA, USA: Association for Computing Machinery, 2005: 340-353. https://doi.org/10.1145/1102120.1102165. DOI: 10.1145/1102120.1102165.

[9] DAVANIAN A, QI Z, QU Y, et al. DECAF++: Elastic Whole-System Dynamic Taint Analysis[C/OL]//22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019). Chaoyang District, Beijing: USENIX Association, 2019: 31-45. https://www.usenix.org/conference/raid2019/presentation/davanian.

[10] KANG M G, MCCAMANT S, POOSANKAM P, et al. Dta++: dynamic taint analysis with targeted control-flow propagation.[C]//NDSS. [S.l. : s.n.], 2011.

[11] CLAUSE J, LI W, ORSO A. Dytan: A Generic Dynamic Taint Analysis Framework[C/OL] //ISSTA '07: Proceedings of the 2007 International Symposium on Software Testing and Analysis. London, United Kingdom: Association for Computing Machinery, 2007: 196-206. https://doi.org/10.1145/1273463.1273490. DOI: 10.1145/1273463.1273490.

[12] SZEKERES L, PAYER M, WEI T, et al. SoK: Eternal War in Memory[C]//2013 IEEE Symposium on Security and Privacy. [S.l. : s.n.], 2013: 48-62. DOI: 10.1109/SP.2013.13.

[13] OH N, SHIRVANI P, MCCLUSKEY E. Control-flow checking by software signatures[J]. IEEE Transactions on Reliability, 2002, 51(1): 111-122. DOI: 10.1109/24.994926.

[14] CHECKOWAY S, DAVI L, DMITRIENKO A, et al. Return-Oriented Programming without Returns[C/OL]//KEROMYTIS A, SHMATIKOV V. Proceedings of CCS 2010. [S.l.]: ACM Press, 2010: 559-572. https://checkoway.net/papers/noret_ccs2010.

[15] JOHNSON E, THIEN D, ALHESSI Y, et al. Доверя́й, но проверя́й: SFI safety for native-compiled Wasm[C]//. [S.l. : s.n.], 2021. DOI: 10.14722/ndss.2021.24078.

[16] BIENIA C. Benchmarking Modern Multiprocessors[D]. Princeton University, 2011.

[17] HAAS A, ROSSBERG A, SCHUFF D L, et al. Bringing the Web up to Speed with WebAssembly[J/OL]. SIGPLAN Not., 2017, 52(6): 185-200. https://doi.org/10.1145/3140587.3062363. DOI: 10.1145/3140587.3062363.

[18] WAGNER L. WebAssembly consensus and end of Browser Preview[C/OL]//. [S.l. : s.n.], 2017. https://lists.w3.org/Archives/Public/public-webassembly/2017Feb/0002.html.

[19] FU W, LIN R, INGE D. TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly[J]. ArXiv, 2018, abs/1802.01050.

[20] LEHMANN D, KINDER J, PRADEL M. Everything Old is New Again: Binary Security of WebAssembly[C/OL]//29th USENIX Security Symposium (USENIX Security 20). [S.l.]: USENIX Association, 2020: 217-234. https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann.

[21] LEHMANN D, PRADEL M. Wasabi: A Framework for Dynamically Analyzing WebAssembly[C/OL]//ASPLOS '19: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. Providence, RI, USA: Association for Computing Machinery, 2019: 1045-1058. https://doi.org/10.1145/3297858.3304068. DOI: 10.1145/3297858.3304068.

Acknowledgements

First of all, I would like to express my deepest thanks to my academic supervisor-Professor Yinqian Zhang. Without his great help and encourage, I certainly cannot finish my graduate project so smoothly.

During this project, many seniors and junior in our laboratory gave me a lot of assistant. I would also like to thank Weili Wang, Wei Peng and Shengqi Liu from Southern University of Science and Technology(SUSTech). They shared a lot of ideas with me, which help me a lot.

Finally, I would like to thank all the professors who imparted knowledge to me in SUSTech. I have experienced enough and happy four years in SUSTech.