Universidade da Beira Interior
Departamento de Informática

Nº 162 - 2021:*WebAssembly*

Elaborado por:

Pedro Manuel Pires Lopes

Orientador:

Professor Doutor Paul Crocker

7 de Abril de 2023

Agradecimentos

A conclusão deste trabalho, bem como da grande parte da minha vida académica, não seria possível sem a ajuda dos meus pais e dos meus professores que me orientaram a seguir este caminho. Estes que hoje permitem estar a frequentar esta universidade que me acolheu de braços abertos e que me permitiu estar a desenvolver este projeto.

Este trabalho que me mostrou um novo conjunto de temas que não conhecia e que aumentou ainda mais o meu interesse pelo curso que frequento e pela área que estudo.

A todos estes que eu referi agradeço por fazerem de mim uma pessoa mais culta, obrigado.

Conteúdo

Conteúdo **iii**

Lista de Figuras **vii**

1 Introdução **1**
1.1 Enquadramento 1
1.2 Motivação 1
1.3 Documentos do projeto 1
1.4 Objetivos 1
1.5 Organização do Documento 2

2 Tecnologias e Ferramentas Utilizadas **3**
2.1 Introdução 3
2.2 *C* 3
2.3 *Rust* 3
2.4 *Python* 4
2.5 *Javascript* 4
2.6 *WebAssembly* 4
2.7 *Emscripten* 5
2.8 Conclusões 5

3 Produção e execução de *WebAssembly* **7**
3.1 Introdução 7
3.2 Instalação das ferramentas Base 7
3.2.1 Compilador de C 7
3.2.2 Compilador de Rust 8
3.3 Produção e execução através de um *Browser* 8
3.3.1 Instalar o *Emscripten* 8
3.3.2 Compilar com o *Emscripten* 9
3.3.3 Compilar e executar o *WebAssembly* 9
3.4 Acesso ao Sistema Operativo através do *WebAssembly* 11
3.4.1 The WebAssembly System Interface *(WASI)* 11
3.4.1.1 Compilar para *WebAssembly* através do *WASI* 12

3.4.1.2 Conclusões a cerca do *WASI* 13
3.5 *Runtimes* de *WebAssembly* independentes de *Javascript* 13
3.5.1 *Wasmtime* 13
3.5.2 *Wasmer* 16
3.6 The WebAssembly Binary Toolkit *(WABT)* 19
3.6.1 *Exemplo de uso do* WABT 19
3.6.2 Conclusões do *WABT* 21
3.7 Conclusões 22

4 Análise de Código em *WebAssembly* 23
4.1 Introdução 23
4.2 Análise estática e análise dinâmica 23
4.2.1 Análise estática 23
4.2.2 Análise dinâmica 24
4.3 Análise de *WebAssembly* 24
4.3.1 Instalação das Ferramentas 24
4.3.1.1 Instalar o WebAssembly analysis using binary instrumentation *(WASABI)* 24
4.3.2 *WASABI* 24
4.3.3 Analisar com o *WASABI* 25
4.3.4 Razões para usar o Wasabi 28
4.4 Conclusões 29

5 Análise de Performance 31
5.1 Introdução 31
5.2 Preparação dos testes 31
5.2.1 Compilador do *WebAssembly* 32
5.2.2 Ambiente dos testes 32
5.2.3 *Script* de execução dos testes 32
5.3 Testes de performance 33
5.3.1 Testes nas bibliotecas de *AES-128* 34
5.3.1.1 *C* e *C-WASM* 34
5.3.1.2 *Rust* e *Rust-WASM* 35
5.3.1.3 *C-WASM, Rust-WASM* e *Javascript* 35
5.3.1.4 Conclusões das implementações de *AES-128* . 36
5.3.2 Testes em *RSA-1024* 36
5.3.2.1 *C* e *C-WASM* 37
5.3.2.2 *C-WASM* e *Javascript* 37
5.3.2.3 Conclusões das implementações de *RSA-1024* 38
5.4 Conclusões dos testes de performance 38

6 Conclusões e Trabalho Futuro **41**
6.1 Discussão e Conclusões 41
6.2 Trabalho Futuro 43

Bibliografia **45**

Lista de Figuras

3.1 Diagrama da ferramenta *Emscripten* para a World Wide Web *(WEB)* 11
3.2 Fluxograma da ferramenta *WASI* com o *Wasmtime* 16
3.3 Inspeção de um ficheiro *WebAssembly* compilado pelo *Emscripten* através do *Wasmer* 18
3.4 Inspeção de um ficheiro *WebAssembly* compilado pelo *WASI* através do *Wasmer* 18
3.5 Execução de um ficheiro *WebAssembly* através do *WABT* 20
3.6 Leitura de um ficheiro *WebAssembly* através do *WABT* 21

4.1 Funcionamento da produção da análise através do *WASABI* 25
4.2 Parte do resultado da análise feita através do *WASABI* 28

5.1 Lógica de execução do *script* de análise 33
5.2 Performance obtida da linguagem *C* e da linguagem *Webassembly* em *AES-128* 34
5.3 Performance obtida da linguagem *Rust* e da linguagem *Webassembly* em *AES-128* 35
5.4 Comparação entre o código produzido em *WebAssembly* e *Javascript* em *AES-128* 36
5.5 Comparação entre o código produzido em *C* e *C-WASM* em *RSA-1024* 37
5.6 Comparação entre o código produzido em *WebAssembly* e *Javascript* em *RSA-1024* 38

Lista de Excertos de Código

3.1 Código em C para mostrar uma *string* com 50 inteiros 9
3.2 Segmento de *Javascript* que permite executar as funções criadas. 11
3.3 Trecho de código em *C* que escreve o factorial de um número para um ficheiro chamado *output* 14
3.4 Uma função para multiplicar por dois em *WebAssembly* 19
3.5 leitura do *WebAssembly* em *Javascript* 20
4.1 Extrai o nome de funções 25
4.2 Interpretação e extração de argumentos 26
4.3 Parte de recepção dos *hooks* 27
4.4 *fibonacci* em *C* 27
5.1 Exemplo da execução do *tiny-aes* 33

Acrónimos

WASABI *WebAssembly analysis using binary instrumentation*

API *Application Programming Interface*

DOM *Document Object Model*

GC *Garbage Collector*

UBI Universidade da Beira Interior

WABT *The WebAssembly Binary Toolkit*

WASI *The WebAssembly System Interface*

WEB *World Wide Web*

Capítulo

1

Introdução

1.1 Enquadramento

Este relatório foi feito no contexto da unidade curricular de projeto da Universidade da Beira Interior (UBI), com o sentido de estudo da linguagem *WebAssembly*.

1.2 Motivação

Este trabalho foca-se no estudo e da analise da linguagem *WebAssembly*. Esta linguagem é uma tecnologia recente, em desenvolvimento e compartilha factores em comum com linguagens de baixo nível como *C* e *Rust*. Também irá ser estudado a forma como *WebAssembly* interage com o sistema onde esta opera e o seu desempenho.

1.3 Documentos do projeto

Todos os documentos e códigos associados a este documento encontram-se disponíveis para consulta na plataforma *Github* [1].

1.4 Objetivos

1. Apresentar como se pode compilar e executar binários em *WebAssembly*;

2. Apresentar ferramentas úteis para *WebAssembly* e dar um exemplo de uso delas;

3. Criar um *script* em *Javascript* através da tecnologia WebAssembly analysis using binary instrumentation *(WASABI)* para analisar código em *WebAssembly*;

4. Analisar código gerado para *WebAssembly* e comparar com as suas versões nativas;

5. Comparar os tempos de execução de algoritmos de encriptação entre o *WebAssembly*, as suas implementações nativas e algoritmos semelhantes implementados em *WebAssembly*.

1.5 Organização do Documento

Para que este documento esteja encadeado de forma a explicar o projeto realizado, este encontra-se estruturado da seguinte forma:

1. O primeiro capítulo – **Introdução** – apresenta o projeto, a motivação para a sua escolha, o enquadramento para o mesmo, os seus objetivos e a respectiva organização do documento;

2. O segundo capítulo – **Introdução as Tecnologias Utilizadas** – encontra-se descrito uma pequena introdução das tecnologias utilizadas ao longo do projeto;

3. O terceiro capítulo – **Produção e execução de *WebAssembly*** – encontra-se explicado como se compila e executa *WebAssembly*. Também são apresentadas algumas ferramentas úteis para a linguagem;

4. O quarto capítulo – **Analise de Código em *Webassembly*** – aqui se descreve o funcionamento detalhado do *WASABI* e do o *script* de analise do *WASABI* perante o código *WebAssembly* gerado pelo o utilizador;

5. O quinto capítulo – **Analise de Performance** – é feito um conjunto de testes em bibliotecas de criptografia e é comparado entre linguagens com alta performance como *C* e *Rust*, os seus compilados em *WebAssembly* e comparado com bibliotecas de *Javascript*;

6. O sexto capítulo – **Conclusões e Trabalho Futuro** – é apresentado algumas limitações e alguns avanços na linguagem *WebAssembly*, são referidos algumas breves conclusões e apresentados algumas situações que poderiam ser feitas no futuro.

Capítulo

2

Tecnologias e Ferramentas Utilizadas

2.1 Introdução

Neste capítulo é feita uma pequena introdução de algumas tecnologias que foram usadas durante este projeto, as tecnologias que não foram apresentadas aqui irão ser apresentadas nos capítulos seguintes.

2.2 *C*

A linguagem de programação C, uma linguagem de baixo nível, permite com que sejam criados programas que executam com um dos maiores níveis de eficiência por permitir uma gestão manual dos recursos do computador ao programador, por esse facto esta linguagem torna-se um alvo útil para o teste de falhas nessa gestão. Desta forma é possível gerar de forma fácil código com falhas como por exemplo os *Memory leaks* onde se reservam memoria do sistema para uma ou mais variáveis, mas onde esse programa não chega a libertar a memoria reservada.

Por outro lado a linguagem *C* sofre a nível de segurança por permitir ao utilizador que este execute código com falhas que possam ser exploradas.

2.3 *Rust*

A linguagem de programação *Rust*, uma linguagem de baixo nível, criada recente com o intuito de gerar código seguro e eficiente em níveis de *perfor-*

mance, que tenta garantir que o código gerado pelo programador seja seguro e que as falhas na memoria sejam seguras[2] e que estas não afectem o utilizador final.

Esta linguagem torna-se útil porque permite que sejam comparados programas de semelhante performance, mas com a garantia de que o código é seguro.

2.4 *Python*

A linguagem de programação *Python*, criada com o intuito de se criar *scripts* de alto nível que permitem implementar algoritmos com grande facilidade apesar do custo pesado no factor da performance.

O *Python* facilitou a forma como foram executados de forma automática os testes e serviu para criar executar um servidor pequeno para alguns testes do *WebAssembly*.

2.5 *Javascript*

A linguagem de programação *Javascript*, criada no intuito de servir a World Wide Web *(WEB)* através de *scripts*, permite aos programadores criar código com um alto nível de abstração mas com um custo enorme no factor da performance e da segurança, apesar que actualmente o *Javascript* use o motor de *Javascript V8*[3] pela *Google* que promove experiência melhorada da *WEB* e de aplicações que usam *Javascript* como o *NodeJs*.

Neste caso *Javascript* irá permitir com que seja possível que através do *WASABI* seja criada uma análise do conteúdo gerado pelo o WebAssembly que originam um conjunto de métodos e de dados que habilitam ao programador criar *scripts* que análisem ao pormenor o que ocorre naquela pagina do *browser*, ou seja, o que ocorre no ficheiro WebAssembly.

2.6 *WebAssembly*

A linguagem de programação WebAssembly permite com que programas que precisem de muitos recursos computacionais possam ser executados na *WEB* com um excelente desempenho em performance quando comparado ao *Javascript*.

Esta é a segunda linguagem que conseguiu unificar os maiores *browsers* disponíveis do lado do cliente, sem contar com os *plugins* como o *Adobe Flash* ou como o *Java* através de *Java-Applets*.

O WebAssembly tem como objectivo de ser executada ao lado de *Javascript*, no sentido em que o *backend* é construído em *WebAssembly* e, por outro lado, o *frontend* é construído em *Javascript*.

Neste caso, iremos usar o *WebAssembly* como alvo de análise para podermos ver o comportamento que ocorre durante a execução.

2.7 *Emscripten*

A ferramenta *Emscripten* permite com que seja possível compilar código originado de *C*, *C++*, *Rust* ou outra qualquer linguagem que consiga compilar para o *LLVM IR*[4], isto é, o *Emscripten* através do código em *LLVM IR*[4] irá produzir *WebAssembly* que possa ser executado na *WEB*, no *NodeJS* ou em qualquer ferramenta que execute *WebAssembly*[5].

No nosso caso esta foi usada para compilar código em *C* e em *Rust* para que fosse possível haver comparações no *WebAssembly* gerado por ambos.

2.8 Conclusões

Com base no que foi apresentado, nota-se um foco ferramentas é possível perceber-se que este projecto foca-se em apresentar a linguagem *WebAssembly* a quem não a conhece e também apresar um conjunto de ferramentas que permitem agilizar o trabalho com esta nova linguagem.

Capítulo

3

Produção e execução de WebAssembly

3.1 Introdução

Neste capitulo será explicado como se pode compilar para *WebAssembly* e algumas ferramentas existentes que permitem executar o binário produzido. Também iram ser apresentadas algumas ferramentas para *debugging* de ficheiros em *WebAssembly*. Em suma, este capitulo da uma breve introdução de como começar a usar a linguagem *WebAssembly* e as ferramentas utilitárias existentes.

3.2 Instalação das ferramentas Base

Vão ser introduzidas compiladores para a linguagem *C*, *Rust* e com base nos dois anteriores serão apresentados compiladores para *WebAssembly* que utilizam os compiladores destes para produzir ficheiros em *WebAssembly*.

3.2.1 Compilador de C

Para instalar um compilador de C, no caso o compilador *Clang* é preciso escrever no terminal:

```
sudo apt-get install clang
```

Assim é possível compilar e executar código de origem C com os comandos:

```
clang -o "nomeDoOutput" "nomeFicheiroC.c"
./nomeDoOutput
```

3.2.2 Compilador de Rust

No caso de *Rust* que é preciso para compilar algumas ferramentas usadas noutros capítulos é preciso fazer os seguintes comandos:

```
sudo apt-get install curl -y
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

source $HOME/.cargo/env
source ~/.profile
```

No caso para executar é preciso:

```
rustc "nomedoFicheiro.rs"
./nomedoFicheiro

#O mais comum é usar o cargo dentro do projeto em rust
cargo run
```

3.3 Produção e execução através de um *Browser*

Uma forma fácil tanto para a linguagem *C* e *Rust* para produzir *WebAssembly* pronto para ser corrido num *browser* é através do compilador *Emscripten*, este permite com que se possa compilar para a linguagem *WebAssembly* sem grande necessidade de adaptar o código produzido.

3.3.1 Instalar o *Emscripten*

Para instalar o *Emscripten* é preciso o *Python* e o *git*:

```
#instalar o git e python3
sudo apt-get update
sudo apt-get install git python3

#instalar o Emscripten
git clone https://github.com/emscripten-core/emsdk.git
cd emsdk
git pull
# Pode escolher entre instalar e ativar a versão mais recente
#./emsdk install|activate latest
#ou uma versão especifica
./emsdk install 1.39.20
```

```
./emsdk activate 1.39.20

#adicionar ao bash_profiles
nano .bash_profile
source ./emsdk/emsdk_env.sh
source ~/.bashrc

#instalar dependências do Emscripten
sudo apt-get install cmake
sudo apt-get install default-jre

#Testar se o Emscripten está a funcionar
emcc -v
```

3.3.2 Compilar com o *Emscripten*

Para usar o compilador do *Emscripten* para produzir *WebAssembly*:

```
#compilar código Rust:
rustc --target=wasm32-unknown-emscripten ".rs" -o ".html"
#ou
cargo build --target=wasm32-unknown-emscripten
#na pasta target/wasm/debug
#adicionar um .html modelo do emscripten

#compilar código C ou C++:
emcc -o "nomeDoOutput.html" "nomeDoFicheiro.c"
#ou
emcc -o "nomeDoOutput.html" "nomeDoFicheiro.cpp"
```

3.3.3 Compilar e executar o *WebAssembly*

Através do *Emscripten* e do auxilio de um servidor local através de *Python* podemos correr e testar o *WebAssembly* produzido. Neste caso para testarmos iremos usar o seguinte trecho de código que mostra para o ecrã um contador de 1-50 alocando um vector de 50 inteiros e depois escrevendo esses 50 inteiros num *array* de caracteres e por fim mostrando esse *array* para o ecrã:

```c
#include <stdio.h>
#include <stdlib.h>
int* vetor(){
    int* vetor = malloc(50*sizeof(int));
```

```c
    for (int i = 1; i<50;vetor[i] = i,i++);
    return vetor;
}
char* count_format_towasmprint(int* vetor){
    char* str = malloc(500 * sizeof(char));
    for(int i = 0; i < 50;i++)
        sprintf(str,"%s%d\n",str,vetor[i]);
    return str;
}
int main(){
    printf("%s\n",count_format_towasmprint(vetor()));
}
```

Excerto de Código 3.1: Código em C para mostrar uma *string* com 50 inteiros

Através do compilador do *Emscripten* usando a *flag* -o com o *output* com o formato *HTML*, este irá produzir os ficheiros em *HTML* em *Javascript* e em *WebAssembly* respectivos do ficheiro em *C* aqui o *Webassembly* está pronto a ser executado por um *Browser* através do ficheiro *HTML* ou pelo *NodeJS* através do ficheiro *Javascript* gerado como demonstrado abaixo:

```
#compilado desta forma recomenda-se o uso de mais flags
#para obter um resultado com uma performance maior
#usar as flags -o3 -ffast-math
emcc test.c -o test.html

#através do Browser recomenda-se o Python http.server
#Este chama através do html chama o .js que chama o .wasm
python3 -m http.server "port"
localhost:"port"

#pelo NodeJs que por sua vez chama o WebAssembly
#pelo ficheiro Javascript
node teste.js
```

Entrando no endereço do servidor local podemos ver uma pagina gerado pelo *Emscripten* e nela uma pequena janela onde é executado o nosso ficheiro *WebAssembly*. Concluindo, a compilação através do *Emscripten* para a *WEB* segue o seguinte diagrama:

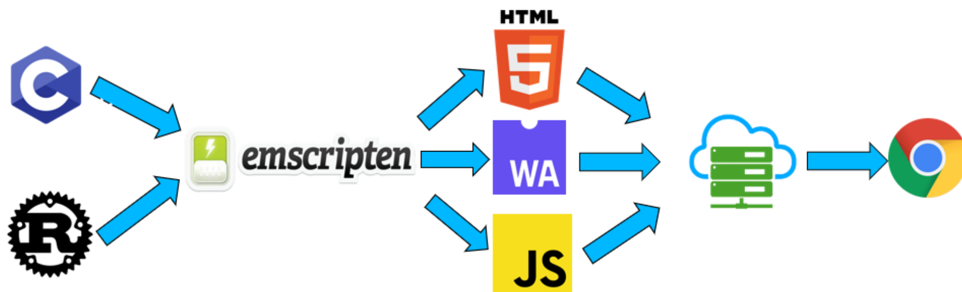

Figura 3.1: Diagrama da ferramenta *Emscripten* para a *WEB*

Caso queira-mos ser independente da plataforma fornecida pelo *Emscripten* podemos chamar diretamente o nosso ficheiro *WebAssembly* através da seguinte instrução:

```
WebAssembly.instantiateStreaming(fetch('nome-do-ficheiro.wasm'),
    importObject)
    .then(results => {
    // Do something with the results!
});
```

Excerto de Código 3.2: Segmento de *Javascript* que permite executar as funções criadas.

3.4 Acesso ao Sistema Operativo através do *WebAssembly*

Iremos usar o sistema implementado pelo The WebAssembly System Interface *(WASI)* para explicar como o *WebAssembly* consegue executar comandos do sistema mesmo não referindo o sistema que irá ser usado para executar o código compilado.

3.4.1 *WASI*

Sem o uso do *WASI* o *WebAssembly* usa o *Javascript* para conseguir executar chamadas ao sistema. Esta situação além de aumentar o tempo de execução do ficheiro binário por recorrer a chamadas pelo lado do *Javascript* que por sua vez requer uma chamada ao *Browser* ou ao *NodeJS* para saber em que ambiente se encontra. No entanto, sem acesso ao *Javascript* não teríamos acesso a essas funcionalidades fornecidas pelo sistema operativo. Para resolver esse problema foi desenvolvido o *WASI*.

O *WASI*[6] é uma Application Programming Interface *(API)* desenhada pelo projecto *Wasmtime*[7] que tem como base fornecer acesso a ferramentas do sistema operativo como *sockets*, *clocks* e *filesystems* entre outras.

O *WASI* através do *Wasmtime* permite executar *WebAssembly* de forma independente dos *browsers* ou do *Javascript*, este implementa também um sistema de *sandbox* como o *WebAssembly* mas permitindo que esse sistema tenha funções de *input/output* independente da plataforma usada.

Atualmente o *WASI* apenas garante suporte para as linguagens *C*, *C++* e *Rust*.

3.4.1.1 Compilar para *WebAssembly* através do *WASI*

Com base no que foi dito anteriormente (3.4.1), irá-se explicar como funciona apenas para estas três línguas, para a linguagem *Rust* apenas é necessário recorrer a duas instruções, que, neste caso, são adicionar o *target* de compilação e referir o mesmo na compilação de um projecto em *Rust*, assim temos as seguintes instruções:

```
#para instalar o wasi
rustup target add wasm32-wasi

#para compilar de rust para wasm
cargo build --target=wasm32-wasi

#ou através do rustc
rustc "nome-do-programa".rs --target=wasm32-wasi -o "nome-final".wasm
```

No caso de *C* ou *C++* é necessário o uso do compilador *Clang*[8], de preferência o *clang* fornecido pelo *wasi-sdk*[9] através do seguinte comando:

```
#vai ser usado a versão 12.0 do wasi-sdk
wget https://github.com/WebAssembly/wasi-sdk/releases/download/
wasi-sdk-12/wasi-sdk-12.0-linux.tar.gz

#para extrair
tar xvf wasi-sdk-12.0-linux.tar.gz

#para compilar de C ou C++ com acesso as standard libraries
wasi-sdk-12.0/bin/./clang "nome-do-ficheiro".c
--sysroot="caminho-ate-ao-sdk"/share/wasi-sysroot/
-o "nome-final".wasm
```

Desta forma podemos compilar o ficheiro nas linguagens *C* ou *C++* para *WebAssembly* sem problemas.

3.4.1.2 Conclusões a cerca do *WASI*

Podemos ver algum foco por parte dos desenvolvedores do *WASI* algum foco em segurança, no sentido em que, o programa que é executado só faz algo ao sistema do utilizador, se lhe for dado acesso pelo *Wasmtime*, infelizmente algumas funcionalidades dos sistemas operativos como *sockets* ainda estão em desenvolvimento e portanto não são possíveis serem usadas. Desta forma podemos concluir que o *WASI* é uma boa alternativa para se usar, visto que este é independente da plataforma que usemos, e que, de forma universal, permite ao programador produzir código independente da plataforma que este usa e dar esse trabalho ao *WASI* para converter para a plataforma que for usada.

3.5 *Runtimes* de *WebAssembly* independentes de *Javascript*

3.5.1 *Wasmtime*

Uma das alternativas para correr o *WebAssembly* produzido *WASI* é utilizar o *Wasmtime*[7], este é um *runtime* para *WebAssembly* que complementa o compilador do *WASI*. Esta ferramenta permite executar directamente *WebAssembly* ou incorporar o código em *WebAssembly* com linguagens como o *C*, *Rust*, *.NET*, *Python*, *GO* e *Bash*.

Para instalar e executar o *Wasmtime* basta simplesmente executar seguintes comandos:

```
#Para instalar basta correr este comando
curl https://wasmtime.dev/install.sh -sSf | bash

#Para executar um ficheiro em WebAssembly com Wasmtime
wasmtime "nome-do-ficheiro".wasm arg1 arg2 arg3
```

A utilidade de usar o *Wasmtime* em conjunto com o *WASI* é o facto de garantirmos que o ficheiro previamente compilado não consegue escrever no sistema sem que o utilizador permita, isto é, sem que o utilizador especifica as directorias que vão ser usadas pelo programa, este nunca irá conseguir ler ou editar, ou criar ficheiros sem que o utilizador previamente lhe permita. Para se perceber esta situação vejamos o seguinte exemplo:

```c
#include <stdio.h>
#include <stdlib.h>
#define uint64 unsigned long long
uint64 factorial_aux(uint64 numero, uint64 counter){
  if(counter == 0) return numero;
  return factorial_aux(counter*numero,counter-1);
}
void factorial(uint64 numero){

  FILE *f = fopen("output.txt","w");

  if (numero < 0) {
    fprintf(f,"Factorial de %lld nao existe\n",numero);
    fclose(f);
    return;
  }
  fprintf(f,"Factorial de %lld = %lld\n",numero,factorial_aux(1,numero))
      ;
  fclose(f);
}

int main(int argc, char **argv){
  factorial(atoi(argv[1]));
}
```

Excerto de Código 3.3: Trecho de código em *C* que escreve o factorial de um número para um ficheiro chamado *output*

Este programa ao ser compilado para *WebAssembly* e depois executado iria escrever o factorial do número introduzido no ficheiro *output.txt*, no entanto se for compilado através do *WASI* ao tentar executar o ficheiro, sem que lhe seja passada a informação da pasta onde este se encontra pelo *Wasmtime* irá causar um erro deste tipo:

```
#Execução que causa o erro
wasmtime teste.wasm 19

#Erro produzido
Error: failed to run main module `teste.wasm`

Caused by:
    0: failed to invoke command default
    1: wasm trap: uninitialized element
       wasm backtrace:
           0: 0x374b - <unknown>!fclose
           1:  0x4e0 - <unknown>!factorial
```

```
            2:  0x555 - <unknown>!main
            3: 0x34f0 - <unknown>!__main_void
            4:  0x574 - <unknown>!__original_main
            5:  0x2cd - <unknown>!_start
note: run with ‘WASMTIME_BACKTRACE_DETAILS=1‘
     environment variable to display more information
```

Este erro acontece devido ao facto do programa estar a ser corrido em *sandbox*, isto é, ele não ter acesso à ou as pastas que o programa irá usar, impedindo, assim, que o programa indevidamente aceda a ficheiros que não são permitidos. Ora para que possamos enviar o nosso resultado basta passar para o *Wasmtime* o seguinte comando:

```
#Execução geral de vários inputs/outputs
#indicando os caminhos
#desde que estes se encontrem na pasta
#do ficheiro em WebAssembly
wasmtime --dir="pasta_1" ... --dir="pasta_n" "fich_1" ... "fich_n" args

#Aplicando o que está referido pelo exemplo geral
wasmtime --dir=. teste.wasm 19

#Resultado do conteúdo do ficheiro output.txt
Factorial de 19 = 121645100408832000
```

Assim, podemos, desta forma, passar ao *Wasmtime* a informação que o ficheiro requer permissões de *input* e ou *output*, desde que estas se encontrem dentro da pasta do ficheiro *WebAssembly*. Obviamente isto limita o que o ficheiro *WebAssembly* no entanto o *Wasmtime* tem capacidade de enviar para outra pasta o resultado obtido através da *flag –mapdir=* com o caminho até a pasta que será enviado o resultado.

Em suma o *Wasmtime* em conjunto do *WASI* pode ser brevemente resumido no seguinte fluxograma:

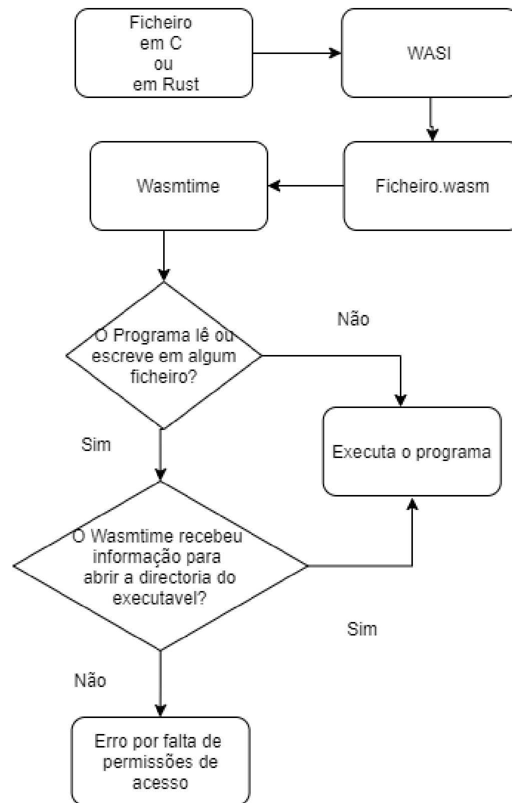

Figura 3.2: Fluxograma da ferramenta *WASI* com o *Wasmtime*

Este diagrama apenas representa um exemplo do que acontece entre o *WASI* e o *Wasmtime* como o exemplo do factorial, no entanto, o *WASI* segue um comportamento semelhante ao representado no diagrama anterior para outras situações.

3.5.2 *Wasmer*

Tal como o *Wasmtime* o *Wasmer*[10] este *runtime* permite executar *WebAssembly* sem a necessidade de um *browser* ou de *Javascript.* Para alem de executar *WebAssembly* este tem algumas capacidades extra como o *inspect* que permite consultar informações de variáveis globais, funções, o tamanho do ficheiro e outros.

Para usar esta ferramenta basta fazer o seguinte comando:

```
#instalar o Wasmer
curl https://get.wasmer.io -sSfL | sh
```

```
#Executar binários com o Wasmer
wasmer run "ficheiro".wasm
```

Ao instalar o *Wasmer* vem acoplado o gestor de pacotes *wapm* que é constituído por ficheiros compilados para *WebAssembly*.

Uma vez que este suporta o *WASI* também temos implementando o sistema que usa *–dir=* para escrever ou ler ficheiros que estejam no sistema, tal como acontece no *Wasmtime*.

Uma das funcionalidades interessantes do *Wasmer* é a sua capacidade de consultar informações sobre o modulo em *WebAssembly*. Para usar essa funcionalidade basta executar o seguinte comando:

```
wasmer inspect "ficheiro".wasm
```

Neste caso para mostrar-mos esta funcionalidade vamos usar o código usado para o *Wasmtime* e iremos compilar o mesmo código através do compilador do *Emscripten*. Utilizando essa ferramenta nesses dois ficheiros em *WebAssembly*:

```
#para o ficheiro compilado pelo Emscripten
wasmer inspect emcc-teste.wasm

#para o ficheiro compilado pelo WASI
wasmer inspect teste.wasm
```

Obtemos por ordem os resultados da inspeção, no primeiro caso através do *Emscripten*:

```
arctumn@LAPTOP-1QNTV8EU:~/Projeto-WASM/wasi/wasi-c$ wasmer inspect emcc-teste.wasm
Type: wasm
Size: 24.6 KB
Imports:
  Functions:
    "wasi_snapshot_preview1"."fd_read": [I32, I32, I32, I32] -> [I32]
    "wasi_snapshot_preview1"."fd_close": [I32] -> [I32]
    "wasi_snapshot_preview1"."fd_seek": [I32, I64, I32, I32] -> [I32]
    "wasi_snapshot_preview1"."fd_write": [I32, I32, I32, I32] -> [I32]
    "wasi_snapshot_preview1"."proc_exit": [I32] -> []
    "wasi_snapshot_preview1"."args_sizes_get": [I32, I32] -> [I32]
    "wasi_snapshot_preview1"."args_get": [I32, I32] -> [I32]
  Memories:
  Tables:
  Globals:
Exports:
  Functions:
    "__wasm_call_ctors": [] -> []
    "_start": [] -> []
    "__errno_location": [] -> [I32]
    "fflush": [I32] -> [I32]
    "free": [I32] -> []
    "malloc": [I32] -> [I32]
    "stackSave": [] -> [I32]
    "stackRestore": [I32] -> []
    "stackAlloc": [I32] -> [I32]
    "__set_stack_limit": [I32] -> []
    "__growWasmMemory": [I32] -> [I32]
  Memories:
    "memory": not shared (256 pages..256 pages)
  Tables:
  Globals:
    "__data_end": I32 (constant)
```

Figura 3.3: Inspeção de um ficheiro *WebAssembly* compilado pelo *Emscripten* através do *Wasmer*

Apesar do facto que o *Emscripten* usar as próprias funções para versão da *WEB*, que requer auxílio ao *Javascript*, a versão independente do *browser* encontra-se a usar o *WASI* para aceder ao sistema de ficheiros do sistema operativo.

Para o ficheiro compilado através do *WASI* obtemos o resultado seguinte:

```
arctumn@LAPTOP-1QNTV8EU:~/Projeto-WASM/wasi/wasi-c$ wasmer inspect teste.wasm
Type: wasm
Size: 47.7 KB
Imports:
  Functions:
    "wasi_snapshot_preview1"."proc_exit": [I32] -> []
    "wasi_snapshot_preview1"."args_sizes_get": [I32, I32] -> [I32]
    "wasi_snapshot_preview1"."args_get": [I32, I32] -> [I32]
    "wasi_snapshot_preview1"."fd_write": [I32, I32, I32, I32] -> [I32]
    "wasi_snapshot_preview1"."fd_seek": [I32, I64, I32, I32] -> [I32]
    "wasi_snapshot_preview1"."fd_close": [I32] -> [I32]
    "wasi_snapshot_preview1"."fd_prestat_get": [I32, I32] -> [I32]
    "wasi_snapshot_preview1"."fd_prestat_dir_name": [I32, I32, I32] -> [I32]
    "wasi_snapshot_preview1"."fd_fdstat_get": [I32, I32] -> [I32]
    "wasi_snapshot_preview1"."path_open": [I32, I32, I32, I32, I32, I64, I64, I32, I32] -> [I32]
    "wasi_snapshot_preview1"."fd_fdstat_set_flags": [I32, I32] -> [I32]
    "wasi_snapshot_preview1"."fd_read": [I32, I32, I32, I32] -> [I32]
  Memories:
  Tables:
  Globals:
Exports:
  Functions:
    "_start": [] -> []
  Memories:
    "memory": not shared (2 pages..)
  Tables:
  Globals:
```

Figura 3.4: Inspeção de um ficheiro *WebAssembly* compilado pelo *WASI* através do *Wasmer*

Após ser inspecionado o ficheiro compilado pelo *WASI* à priori encontramos o que se esperava obter, no entanto, encontra-se peculiar o facto que

ambos os ficheiros utilizam em grande parte as mesmas funções, o ficheiro compilado através do *WASI* ocupa quase o dobro do ficheiro do *Emscripten*.

Assim para além de um *runtime* o *Wasmer* também permite consultar rapidamente algumas das funções, variáveis globais (ou constantes) e outros usados pelos respectivos compiladores.

Em suma o *Wasmer* é uma alternativa bastante capaz ao *Wasmtime* para executar de forma independente *WebAssembly*, ou seja, sem auxilio do *browser*.

3.6 The WebAssembly Binary Toolkit *(WABT)*

O *WABT*[11] é um conjunto de ferramentas criadas na linguagem *C* destinadas para *debug* de ficheiros em *WebAssembly*. Estas ferramentas permitem analisar, compilar e executar e forma desenhadas de forma a existir uma fácil integração com outras ferramentas. Este conjunto de ferramentas podem ser instaladas através do compilador *Clang* ou pelo terminal do *Ubunto* através do comando:

```
sudo apt-get install wabt
```

3.6.1 *Exemplo de uso do* WABT

Para testar-se o funcionamento foi usada um trecho de código escrito em *WebAssembly* para teste da ferramenta *WABT*:

```
(module
  (func $mulby2 (param f32) (result f32) ;; nome_funcao param_in
      param_out
    local.get 0 ;; parametro 0
    f32.const 2
    f32.mul ;; multiplica os elementos atras mul apenas aceita dois
        argumentos
  )
  (func (export "output") (result f32) ;; export para JS param_out
    f32.const 5
    call $mulby2 ;; multiplica 5 por 2
   )
)
```

Excerto de Código 3.4: Uma função para multiplicar por dois em *WebAssembly*

Como explicado nos comentários do trecho anterior, a função *output* multiplica o valor cinco pelo valor dois, assim temos um pequeno código de *WebAssembly* em formato de texto[12] agora com a ferramenta *wat2wasm* dispo-

nibilizada pelo *WABT* pode-se produzir um ficheiro em formato binário[13] através do seguinte comando:

```
wat2wasm "nome-fich-anterior".wat -o "nome-fich-final".wasm
```

Desta forma temos a conversão do ficheiro em texto para o ficheiro binário que pode ser lido por um ficheiro *Javascript* através da seguinte forma:

```
const wasmInstance =
      new WebAssembly.Instance(wasmModule, {});
const { output } = wasmInstance.exports;
console.log(output());
```

Excerto de Código 3.5: leitura do *WebAssembly* em *Javascript*

Ou pode ser lido através do programa *wasm-interp* obtendo o seguinte resultado:

```
#comando usado para correr o ficheiro test.wasm
wasm-interp test.wasm --run

#resultado obtido:
output() => f32:10.000000
```

Assim o exemplo acima pode ser demonstrado pelo seguinte diagrama:

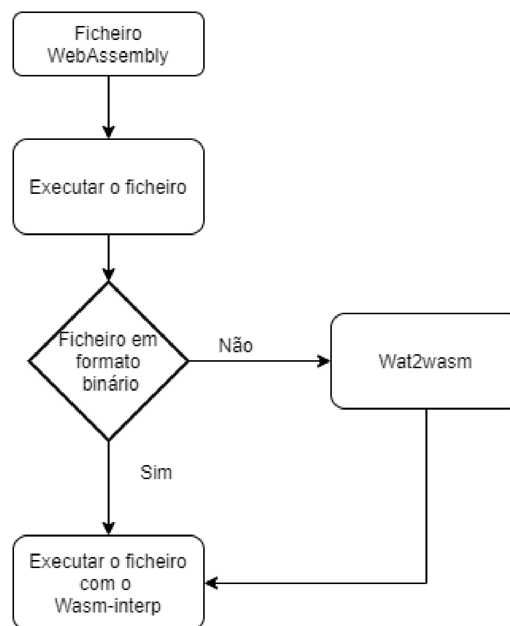

Figura 3.5: Execução de um ficheiro *WebAssembly* através do *WABT*

Por outro lado caso já tenhamos o ficheiro *WebAssembly* em formato binário e queira-mos ler o *WebAssembly* gerado podemos usar a seguinte ferramenta *wasm2wat* com o seguinte comando:

```
wasm2wat "wasmbin".wasm -o "wasmtext".wat
```

Que se pode resumir no seguinte diagrama que demonstra o processo de leitura anterior:

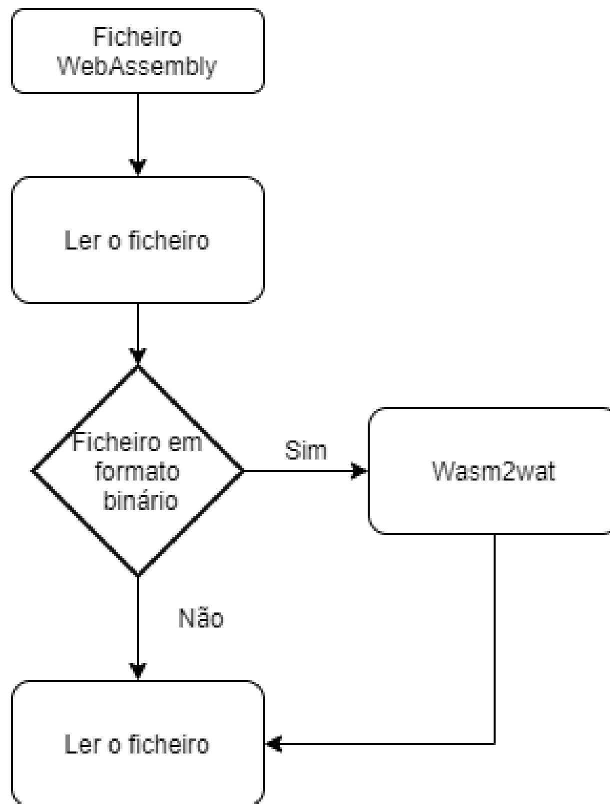

Figura 3.6: Leitura de um ficheiro *WebAssembly* através do *WABT*

3.6.2 Conclusões do *WABT*

Apesar de apenas terem sido mostradas três funcionalidades do *WABT* existem muitas mais e mesmo aquelas mostradas têm outras *flags* úteis para as usar. Por fim o *WABT* é um conjunto de ferramentas úteis para quem precisa manipular ficheiros em *WebAssembly*.

3.7 Conclusões

Ao longo deste capítulo explicou-se como se produz, executa ou se testa *WebAssembly* através de diversas tecnologias como o *Emscripten*, o *WASI* o *Wasmtime* ou o conjunto de ferramentas de *debugging* *WABT*. Apesar de apenas estar referidas algumas formas de compilar para *WebAssembly*, existem ferramentas como o *AssemblyScript*, uma linguagem semelhante ao *TypeScript*, como o *Pyodide* para *Python* compilando o seu interpretador para *WebAssembly* e também referido neste capítulo na secção relacionada ao *WABT*[11] (3.6) utilizando *S-expressions*[14].

Em suma, este capítulo teve como objetivo de demonstrar uma parte fundamental do *WebAssembly* para que seja possível portar projetos em *Javascript* ou em outras linguagens para uma linguagem que pode ser executada na maior parte dos *browsers*.

Capítulo

4

Análise de Código em WebAssembly

4.1 Introdução

Neste capitulo vai se explicar como se pode fazer análises ao código compilado para *WebAssembly*, através da ferramenta *WASABI*.

4.2 Análise estática e análise dinâmica

Os conceitos de análise estática e a análise dinâmica correspondem a modelos diferentes de análise. Enquanto que a análise dinâmica consistem em avaliar e analisar a aplicação durante a sua execução, em contraste a análise estática analisa e avalia a aplicação sem a sua execução através duma análise ao código fonte (ou versão compilada).

4.2.1 Análise estática

A análise estática foca-se em corrigir falhas como erros de programação, violação de *standards* de desenvolvimento, valores indefinidos, vulnerabilidades. Esta também é usada para detectar falhas no código que possam levar a *buffer overflows* ou para detectar falhas que não iriam acontecer sem que passa-se algum tempo na execução de programas.

4.2.2 Análise dinâmica

A análise dinâmica tenta corrigir falhas no comportamento do código, isto é, através do acesso aos valores das variáveis ou do estado do programa. Esta consegue detectar falhas como valores incorrectos, detectar a origem de *Null pointers*, detectar que ciclos demoram mais tempo a ser executados ou erros de memoria que possam corromper o estado do programa.

4.3 Análise de *WebAssembly*

Nesta parte será introduzido a ferramenta *WASABI* e de como se a pode utilizar para fazer uma análise dinâmica ao *WebAssembly*. Neste caso a análise consistirá em extrair os tipos das funções através do modulo que o *WASABI* disponibiliza.

4.3.1 Instalação das Ferramentas

4.3.1.1 Instalar o *WASABI*

Para instalar o *WASABI* é preciso a linguagem *Rust* e da ferramenta *git*:

```
git clone https://github.com/danleh/wasabi
cd wasabi
cargo install --path .
```

Para executar o WASABI é preciso:

```
wasabi --output-dir "nomeDaPasta" "ficheiro.wasm"
```

4.3.2 *WASABI*

A ferramenta *WASABI* é um programa que serve para análise dinâmica como o *Valgrind*[15], mas neste caso redireccionado para código em *WebAssembly*.

O *WASABI* foi feito em *Rust* e usa a linguagem *Javascript* como intermediário para a produção de análises.

O *WASABI* é composto por duas fases. A primeira fase consistem em adicionar instruções ao *WebAssembly* gerado que permitem ao *script* do *WASABI* produzir um modulo com as informações do nosso binário. A segunda fase consiste em aplicar uma análise dinâmica do programa gerado pelo *WebAssembly* através de *scripts* que trabalham sobre as instruções extraídas através do ficheiro *Javascript* gerado pelo *WASABI*[16].

4.3.3 Analisar com o *WASABI*

Para fazer uma análise do ficheiro em *WebAssembly* através da ferramenta *WASABI* iremos usar o *Browser* para servir de intermediário pois iremos gerar mais um ficheiro em *Javascript* para o teste. Assim, iremos seguir o seguinte diagrama:

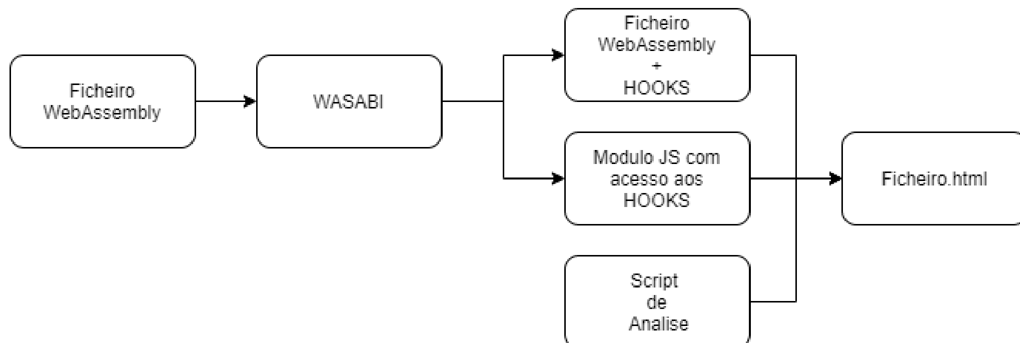

Figura 4.1: Funcionamento da produção da análise através do *WASABI*

Assim para começar iremos fazer os comandos:

```
#escolhemos a pasta atual para uso pratico
wasabi --output-dir=. test.wasm
```

Este comando acima gera um novo *WebAssembly* que tem (segundo os desenvolvedores da ferramenta) os *HOOKS* com a informação sobre todo o ficheiro *WebAssembly* gerado através do *Emscripten*.

Para que se possa fazer uma análise é preciso criar um *script* auxiliar em *Javascript* que através de um modulo, vai ser responsável por orquestrar como a informação capturada é utilizada, no caso iremos usar um *script* que produz uma lista de *strings* com todas as assinaturas das chamadas as funções no ficheiro em *WebAssembly*.

Foi criada uma função que pega no numero identificador de uma outra função e que lhe extrai o identificador, caso esta não seja originada no *WebAssembly*.

```
function fctName(fctId) {
        const fct = Wasabi.module.info.functions[fctId];
        if (fct.export[0] !== undefined) return fct.export[0];
        if (fct.import !== null) return  fct.import;
        //caso falhe a procura do nome, esta retorna o id
        return fctId;
```

```
    }
```

Excerto de Código 4.1: Extrai o nome de funções

Para que se possa extrair e entender o tipo que representa cada argumento da função chamada foram criadas duas funções uma que extrai os argumentos da função e outra interpreta o conteúdo extraído.

```
//Intrepreta os argumentos
function parseType(serializedType){
      if (serializedType === "") return "Received void, Returned void"

      parsedString = "Received args: "

      //Parsing if the input is empty and if is not empty
      if (serializedType.charAt(0) === '|') parsedString += "void"

      serializedType.split('').forEach(function(letter)  {
              switch (letter) {
                  case 'i': //int32
                      parsedString = parsedString + "i32 "
                      break;
                  case 'I': //int64
                      parsedString = parsedString + "i64 "
                      break;
                  case 'f': //float32
                      parsedString = parsedString + "f32 "
                      break;
                  case 'F': //float64
                      parsedString = parsedString + "f64 "
                      break;
                  case '|': //separador entre o que recebe e o que
                      envia
                      parsedString = parsedString + "; Returning args:
                          "
                      break;
                  default:
                      break;
              }
          })

      if (serializedType.charAt(serializedType.length - 1) === '|')
          parsedString += "void"

      return parsedString
      }

      //Extrai os argumentos
      function argType(fctId){
          const fct = Wasabi.module.info.functions[fctId];
```

```
            return parseType(fct.type)
        }
```

Excerto de Código 4.2: Interpretação e extração de argumentos

Para que seja possível receber o que estamos a extrair e a interpretar é preciso chamar o método *call-pre* da seguinte forma:

```
Wasabi.analysis = {
        call_pre(location, targetFunc, _, _) {
            const caller = fctName(location.func);
            const callee = fctName(targetFunc);
            const argcallee = argType(targetFunc);
            listOfElements.push("The function "+ caller
                + " is using the function "
                + callee + "("+argcallee+")");
        },
    };
```

Excerto de Código 4.3: Parte de recepção dos *hooks*

Com este código podemos mostrar ou extrair para um ficheiro todas as funções chamadas dentro do ficheiro *WebAssembly* gerado simplesmente adicionando ao ficheiro *html* uma referencia para o nosso *javascript*.

Para melhor perceber como funciona este *script* ira ser aplicado no seguinte programa em *C* que calcula recursivamente o elemento dez da sequência de *fibonacci*:

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

int fib(n){
  if(n == 0) return  0; else if (n == 1) return 1;
  else return (fib(n-1)+fib(n-2));
}
int main(void){
  printf("output = %d\n",fib(10));
  return 0;
}
```

Excerto de Código 4.4: *fibonacci* em *C*

Através do compilador do *Emscripten* podemos gerar um ficheiro em *WebAssembly* que pode ser usado pelo *WASABI*, produzindo um novo *WebAssembly* e um ficheiro com as referencias para o nosso *script*, por fim adicionando uma referencia para esses dois novos ficheiros no ficheiro *html* basta executar por um *browser*. Para tal iremos usar o *Python* para criar um servidor simples.

```
#Emscripten
emcc fib.c -o3 -ffast-math -o fib.html
```

```
#Adiciona as instruções para análise
wasabi --output-dir=. fib.wasm

#Python
python3 -m http.server
```

Recorrendo ao terminal disponível pelo *browser* chamando a função *write* com o parâmetro *listOfElements* obtemos um ficheiro de texto com a informação que o nosso *script* obteve.

```
function call:
 The function main is using the function 8(Received args: void; Returning args: i32 )
function call:
 The function 8 is using the function 7(Received args: i32 ; Returning args: i32 )
function call:
 The function 7 is using the function 7(Received args: i32 ; Returning args: i32 )
function call:
 The function 7 is using the function 7(Received args: i32 ; Returning args: i32 )
function call:
 The function 7 is using the function 7(Received args: i32 ; Returning args: i32 )
function call:
 The function 7 is using the function 7(Received args: i32 ; Returning args: i32 )
function call:
 The function 7 is using the function 7(Received args: i32 ; Returning args: i32 )
function call:
 The function 7 is using the function 7(Received args: i32 ; Returning args: i32 )
function call:
 The function 7 is using the function 7(Received args: i32 ; Returning args: i32 )
function call:
 The function 7 is using the function 7(Received args: i32 ; Returning args: i32 )
```

Figura 4.2: Parte do resultado da análise feita através do *WASABI*

Como podemos ver o nosso *script* pegou por ordem de chamada o nome das funções e os seus argumentos. Também se pode ver parte da recursividade a acontecer no ficheiro *WebAssembly* através da função com o nome sete que representa a nossa função *fib*.

4.3.4 Razões para usar o Wasabi

Mas qual seria a utilidade de usar uma ferramenta como esta em vez de ler o código-fonte?

Nem sempre se tem acesso ao código-fonte de forma fácil, mesmo com ferramentas para descompilar o código em *WebAssembly*, nem sempre é fácil o interpretar, uma vez que a linguagem deste, usa uma sintaxe que não é tão

legível em comparação ás linguagens de alto nível como *Python* ou *Javascript*, ou até mesmo como linguagens de baixo nível como *C* ou *Rust*.

Por outro lado também é possível analisar as famosas *crypto-miners* por exemplo, através da quantidade de instruções que estas executam e pelo tipo que estas usam, como é feito através do documento *SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks* onde são analisadas varias *crypto-miners* que usam *WebAssembly*[17].

Também é possível fazer uma análise do *heap*, através de uma forma semelhante a usada para extrair a assinatura de funções, é possível procurar por funções de alojamento de memoria como o *malloc* e desalojamento de memória como o *free*, por exemplo ser-lo-ia possível contar a quantidade de vezes que o *malloc* e o *free* foram chamados e descobrir se existe memoria que não foi devidamente libertada no programa, sem nunca se descompilar o ficheiro em *WebAssembly*.

4.4 Conclusões

Apesar de só se referir a ferramenta *WASABI* para análise de *WebAssembly* já existem outras ferramentas que estão a ser adaptadas para esta linguagem como a ferramenta *Manticore*[18] que permite fazer uma a análise semelhante ao *WASABI* mas através da linguagem *Python*, ou através da ferramenta *twiggy*[19] que se foca em fazer uma análise estática aos ficheiros em *Rust* antes de serem compilados para *WebAssembly*.

Em suma já existem um conjunto de ferramentas apesar de serem um pouco primitivas comparando com ferramentas como o *Valgrind*, mas que no entanto permitem extrair e analisar informações dos ficheiros em *WebAssembly* como se pode ver no fim deste capitulo.

Capítulo

5

Análise de Performance

5.1 Introdução

Atualmente alguns dos estudos existentes que avalizaram a performance de *WebAssembly* afirmaram uma diferença de apenas cerca de dez porcento, no entanto foi feito um estudo com o nome de *Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code*[20] onde demonstram que a diferença cima referida apenas se refere para casos muitos específicos, onde o código tem menos de cem linhas, e que segundo o estudo feito na realidade a diferença de performance, caso não se encontre nas circunstancias perfeitas, a performance do *WebAssembly* em relação ao código nativo encontra-se na casa dos quarenta e cinco porcento a cinquenta e cinco porcento mais lento dependendo do *Browser* usado.

Portanto, neste capítulo, será analisada a performance onde o código compilado tem mais de cem linhas, utilizando um caso de estudo diferente do acima referido[20], neste caso serão usadas algumas bibliotecas de criptografia nas linguagens *C*, *Rust* e *Javascript*.

Estas excepto o *Javascript*, serão comparadas as suas versões implementadas em *WebAssembly*, por outro lado o *Javascript* irá ser usado para comparar a suas implementações destas bibliotecas com o *WebAssembly* gerado da linguagem *C* e *Rust* para se comparar a performance de execução.

5.2 Preparação dos testes

Irá ser introduzido as ferramentas que foram usadas, o ambiente em que foram feitos os testes e por fim, será feita uma breve explicação de como funciona o *script* de *Python* responsável pela execução dos testes.

5.2.1 Compilador do *WebAssembly*

Para que seja possível a comparação dos testes, foi usado o compilador *emcc* para produzir o ficheiro binário do *WebAssembly*, dos programas escritos em *C* e em *Rust*, este adiciona algumas instruções extra ao programa feito, que permitem algumas funcionalidades extra como por exemplo enviar argumentos para o programa em *WebAssembly*.

5.2.2 Ambiente dos testes

Para analisar os ficheiros binários em *WebAssembly* e em *Javascript* foi usado o interpretador *NodeJs v12.18.1*, uma vez que este consegue executar directamente os ficheiros com as extensões *.js e .wasm*, num computador com um processador *Intel i7-10510U* com o sistema operativo *Windows* através do *Windows Subsystem for Linux* com a distribuição *Ubuntu 20.04 LTS*.

É de referir que nem todas as bibliotecas são equivalentes, visto que as bibliotecas usadas são de diferentes desenvolvedores, portanto as comparações entre as linguagens não são comparações directas, excepto a comparação entre as linguagens *C* e o compilado de *C* para *Webassembly* e o mesmo é aplicável para a linguagem *Rust*. No entanto, os resultados obtidos simbolizam uma aproximação aceitável para representar o desempenho.

5.2.3 *Script* de execução dos testes

Para que se mecanizar os testes, foi utilizada a biblioteca *subprocess*[21] que permite ao utilizador, neste caso, correr e extrair informação do *standard output* no caso os tempos que demorou a percorrer cada *benchmark*. Este *script* segue a lógica presente no seguinte diagrama:

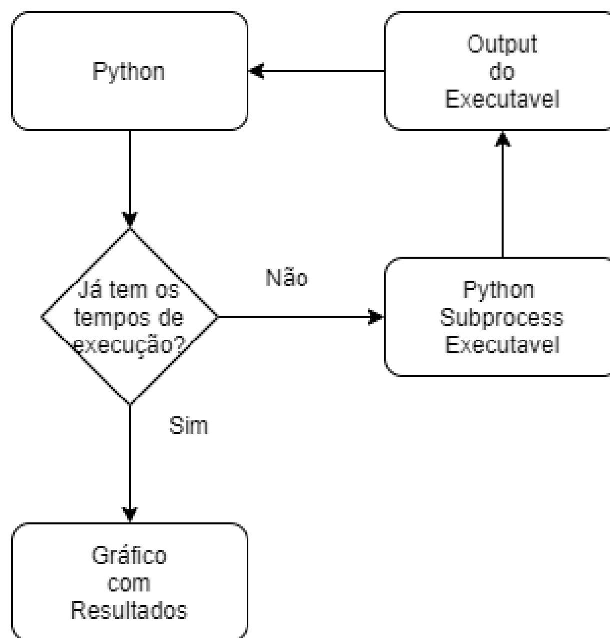

Figura 5.1: Lógica de execução do *script* de análise

Abaixo encontra-se um exemplo do uso da biblioteca *suprocess*[21].

```
import subprocess
result = subprocess.run(
    ['./c-only/tiny-aes/tiny'], #corre este argumento num
        terminal
    stdout=subprocess.PIPE #Extrai o output para o guardar em
        result
    )
#converte o result para ser o tempo calculado em milissegundos
output = float(result.stdout.decode().rstrip('\n'))*1000
#mostra o valor obtido
print(f"Resultado de C -> {output}")
```

Excerto de Código 5.1: Exemplo da execução do *tiny-aes*

Uma vez extraídos, estes são armazenados numa lista que será responsável por manter estes, e que por fim irá ser utilizada para guardar os dados num gráfico, através da biblioteca *matplotlib*[22] que mostra o resultado do tempo demorado com base na linguagem usada.

5.3 Testes de performance

Irão ser analisados os resultados por ordem primeiro os resultados obtidos pelas bibliotecas de *AES-128* e por fim os resultados obtidos das bibliotecas

de *RSA-1024*.

5.3.1 Testes nas bibliotecas de *AES-128*

Para análise da performance foram chamadas cinquenta mil vezes as funções de encriptar e desencriptar, utilizando como alvo de teste uma frase com trezentos e quatro caracteres.

5.3.1.1 *C* e *C-WASM*

Foi feita uma análise com a biblioteca *tiny-aes*[23] e o código produzido através do compilador *emcc*, o ficheiro binário produzido pelo *emcc* obteve uma diferença de aproximadamente quarenta e nove porcento de perda de performance, com uma variação que aumenta a perda de performance por aproximadamente dez porcento, comparado ao código para o ficheiro binário produzido pelo compilador *gcc* resultando na seguinte figura:

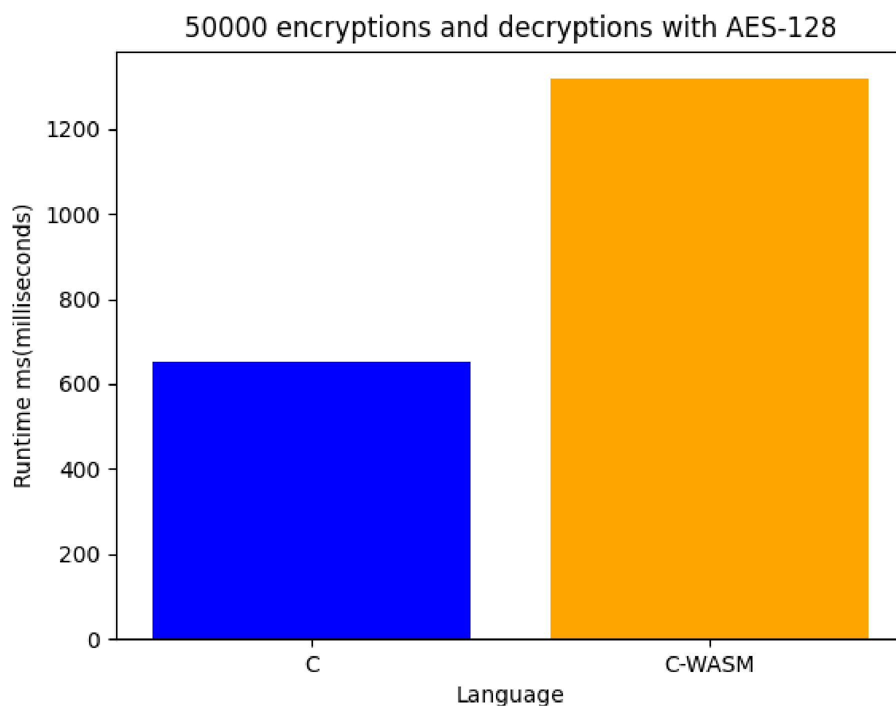

Figura 5.2: Performance obtida da linguagem *C* e da linguagem *Webassembly* em *AES-128*

5.3.1.2 *Rust* e *Rust-WASM*

Como aconteceu no caso anterior, na linguagem *Rust* a biblioteca *AES-128-Rust*[24], tem uma diferença percentual de performance semelhante ao binário compilado pela *tiny-aes*[23], neste caso ocorre uma variação maior que a que ocorria previamente, no entanto há um ganho notável de performance.

Desta forma obteve-se o seguinte gráfico de performance entre o ficheiro binário produzido pela linguagem *Rust* em relação ao ficheiro binário produzido pelo compilador *emcc*:

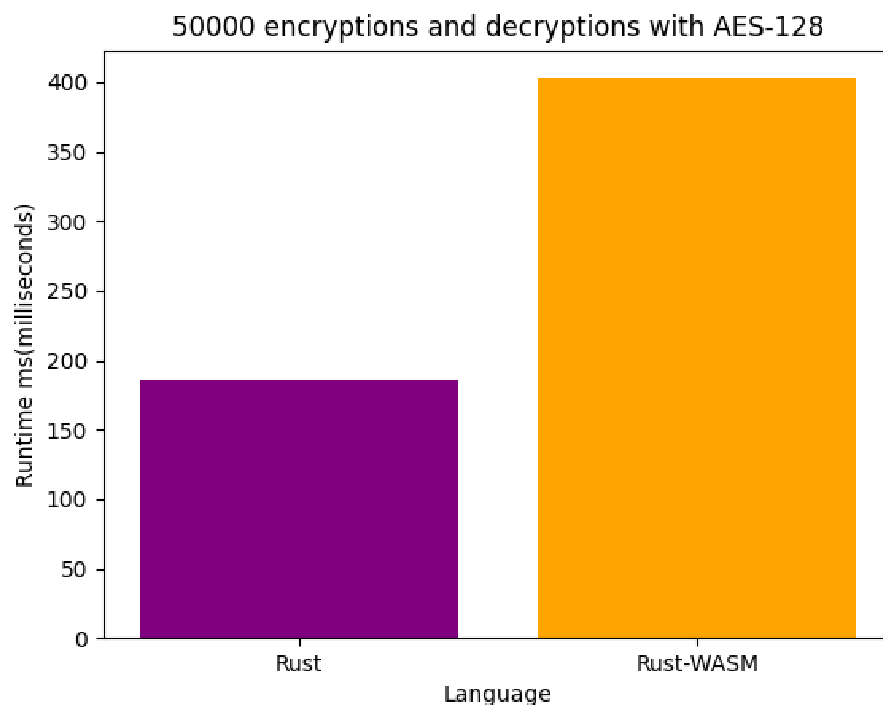

Figura 5.3: Performance obtida da linguagem *Rust* e da linguagem *Webassembly* em *AES-128*

5.3.1.3 *C-WASM, Rust-WASM* e *Javascript*

Neste caso foram comparadas as performance dos vários ficheiros binários com o produzido pelo *javascript* e notou-se o facto que existe um grande impacto de performance, e é muito mais vantajoso usar o código em *WebAssembly*,no caso código escrito em *Rust* e compilado para *WebAssembly*, como podemos ver no gráfico seguinte:

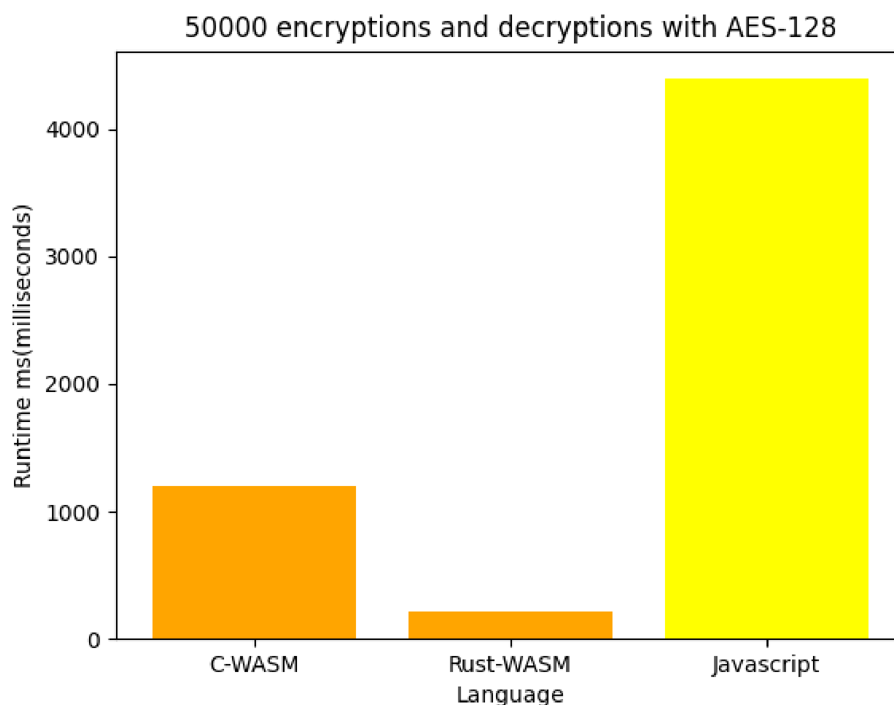

Figura 5.4: Comparação entre o código produzido em *WebAssembly* e *Javascript* em *AES-128*

5.3.1.4 Conclusões das implementações de *AES-128*

Após a observação destes resultados, é notável que existe uma diferença de performance que aumenta conforme o aumento de chamadas a mesma função, dessa forma é mais viável usar uma biblioteca em *WebAssembly* do que recorrer de uma implementação dessa biblioteca em *Javascript*.

5.3.2 Testes em *RSA-1024*

Ao contrário dos testes em *AES-128*, aqui apenas foram chamadas mil vezes as funções de encriptar e desencriptar, visto que os testes demoram muito mais tempo, e um aumento no numero de chamadas a essas funções aumenta exponencialmente o tempo que esses testes demoram. Pela razão anterior referida, foi usa uma frase menor de apenas sessenta e dois caracteres.

Derivado a um *bug* na biblioteca que iria ser usada para testar o código de *Rust* que iria ser compilado para *WebAssembly*, não foram incluídos os testes que iriam representar a linguagem *Rust* nas implementações de *RSA-1024*.

5.3.2.1 *C* e *C-WASM*

Ao contrário do que aconteceu com as implementações de *AES-128*, na implementação usada para este teste[25] houve um resultado mais positivo, onde se notou uma perda de performance de aproximadamente trinta e cinco porcento como se pode ver no seguinte gráfico:

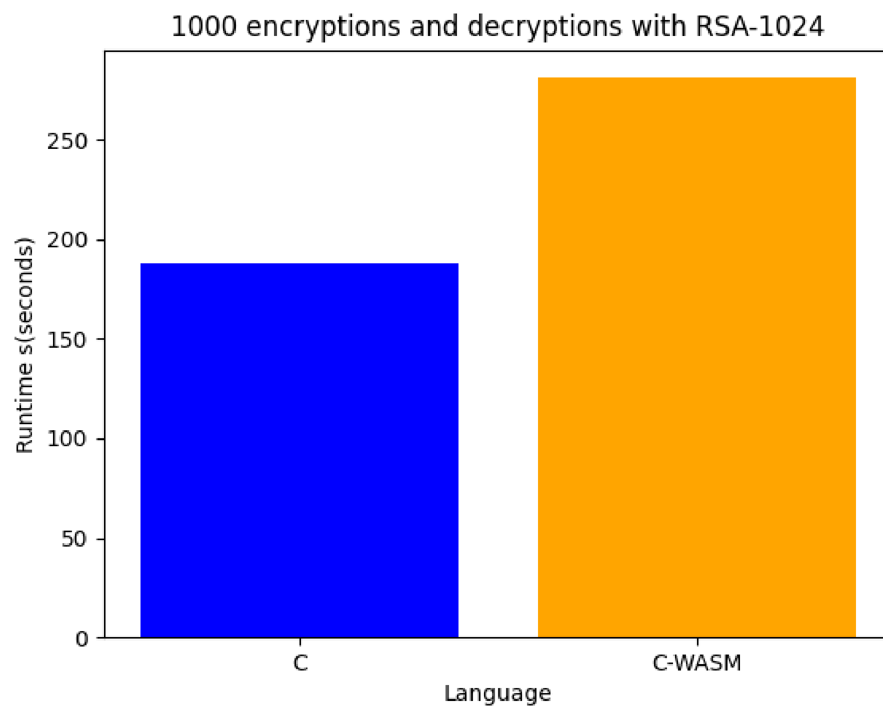

Figura 5.5: Comparação entre o código produzido em *C* e *C-WASM* em *RSA-1024*

5.3.2.2 *C-WASM* e *Javascript*

O mesmo facto que aconteceu com o teste entre *C* e *C-WASM*, repetiu-se aqui, isto é, não existiu uma grande diferença entre a velocidade do ficheiro em *WebAssembly* em relação ao *Javascript*, no entanto num projeto maior onde seja preciso fazer optimizações, é preferível tentar converter esse código para *WebAssembly* como se pode ver na figura seguinte:

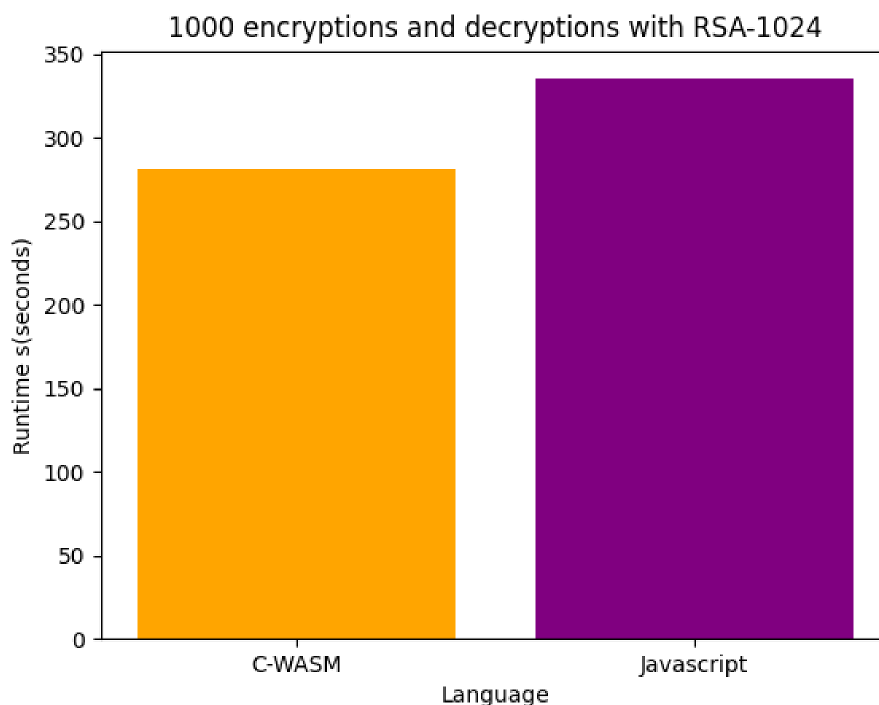

Figura 5.6: Comparação entre o código produzido em *WebAssembly* e *Javascript* em *RSA-1024*

5.3.2.3 Conclusões das implementações de *RSA-1024*

Através dos resultados obtidos nas implementações de *RSA-1024* viu-se que o *WebAssembly* é uma opção válida para aumentar a performance do *Javascript* em casos em que a performance se torne num factor essencial para execução de um grande projeto, obviamente, estes resultados variam bastante porque o *WebAssembly* à priori irá ser executado num *browser*.

5.4 Conclusões dos testes de performance

Apesar destes testes não serem os mais exactos, visto que não são a mesma implementação , é de notar que o *WebAssembly* é ótimo para executar projectos que requerem uma maior eficiência como jogos e editores de imagem ou vídeo.

Como referido no início deste capitulo (5.1) e através dos valores obtidos nos testes, é notável que existe uma grande diferença do valor referido de dez

porcento de diferença e o valor real obtido de quarenta e cinco a sessenta porcento de perda de performance relativamente ao código nativo.

Acredita-se também que com mais avanços nos compiladores para *WebAssembly* a performance se aproxime cada vez mais as implementações originais, tal que, seja preferível incorporar uma mistura de *WebAssembly* com *Javascript* do que apenas *Javascript* e, por fim, garantir que dispositivos que tenham menos recursos sejam capazes de executar confortavelmente projectos que usem muitos recursos no *browser*.

Capítulo

6

Conclusões e Trabalho Futuro

6.1 Discussão e Conclusões

Durante este projeto foi estudada uma tecnologia que se encontra em desenvolvimento, nomeadamente a linguagem *WebAssembly*. Esta que tem como objetivo melhorar a *WEB*, permitindo que algumas aplicações que ficariam fora de um *browser* possam ser executadas nestes. Para que tal possa acontecer o *WebAssembly* contém um leque de ferramentas disponíveis como algumas que foram vistas ao longo deste documento, deste modo a facilitar não só o desenvolvedor que cria as aplicações como também o utilizador final que as utiliza. Todavia, pelo facto de o *WebAssembly* ser uma tecnologia emergente existe ainda algumas limitações que descrevemos em baixo.

- Falta de interação direta com o *Browser*
- Multithreading
- Garbage Collector
- Segurança

No *WebAssembly* que é executado na *WEB* é de notar o facto de que este não tem acesso direto a interações com o *browser*, isto é, o *WebAssembly* não tem acesso direto a tecnologias como o Document Object Model *(DOM)*. Para aceder a estas o *WebAssembly* requer acesso ao *Javascript* para que exista interações entre o programa desenvolvido e do *browser*. Este é um dos casos que ainda não foi resolvido, no entanto como estes existem muitos mais, que à priori irão ser resolvidos.

O que nos leva ao próximo problema o *multithreading*. Muitas das aplicações actuais dependem de ambientes com vários *threads*, como os *autocads*, os jogos e outros. A linguagem por ser ainda muito recente não tem um sistema de *multithreading* completamente implementado, apesar de ferramentas como o *Emscripten*, ou o *wasm-bindgen* em *Rust* terem parcialmente suporte para o fazer, não é garantido que funcione todas as vezes.

Para além do problema acima descrito, começa a surgir um outro problema essencial a muitas linguagens de programação. Hoje em dia é normal termos um Garbage Collector *(GC)* nas novas linguagens, visto que, este traz muitas vantagens por gerir automaticamente a memória do programa pelo o utilizador. Esta situação faz com que compilar para *WebAssembly* seja bastante mais difícil para algumas linguagens, uma vez que estas têm estruturas bastante complexas para serem convertidas. No entanto, actualmente já se encontra um *GC*[26] disponível para *WebAssembly* mas limitado, que de certa forma, permite reduzir o tamanho dos módulos de *WebAssembly*, permite interoperabilidade com outras linguagens, e ter um suporte eficiente a linguagens de alto nível. Este *GC* adiciona um conjunto de instruções disponíveis pela linguagem *WebAssembly*. No fundo, apesar de este estar disponível ainda se encontra em grande parte a ser aprofundado para responder as premissas afirmadas pelos desenvolvedores do *WebAssembly*.

Pelo facto de já começarem a existir algumas soluções aos problemas acima referidos, chegamos a um outro impasse, um problema que afeta todas as linguagens, programadores e utilizadores, isto é, a segurança das aplicações. É preciso garantir que essas soluções para além de cumprirem o que elas prometem, sejam seguras. Hoje em dia não basta as aplicações funcionarem, é preciso que sejam o máximo possível seguras, tanto para os desenvolvedores como para o utilizador final. E apesar deste impasse, os desenvolvedores da linguagem *WebAssembly*, desde o início do desenvolvimento desta, sempre tentam garantir que esta tenda a ser segura para o utilizador final.

Para além do foco inicial dos desenvolvedores do *WebAssembly*, outros grupos com este problema em mente já desenvolveram algumas ferramentas que focam a produção de *WebAssembly* seguro e a execução de código não confiável de forma controlada, sempre com o objetivo da segurança do utilizador final. Neste projecto foram mostradas apenas duas, o *WASI*[6] e o *WasmTime*[7], dentro de um grande conjunto existente. Em suma a comunidade tenta garantir que a linguagem tende a ser o mais seguro possível para a comunidade que desenvolve tecnologias novas e para o utilizador final.

Apesar dos impasses acima referidos, e de ser uma tecnologia nova, o *WebAssembly* consta com alguns grandes avanços, desde os quais se encontra um grande leque de ferramentas e tecnologias que promovem ainda um desenvolvimento ímpar do ecossistema *WebAssembly*.

Através do *WebAssembly* será possível unificar a compilação de projetos para varias plataformas usando o mesmo código para todas as plataformas, sempre com a premissa de garantir que a linguagem *WebAssembly* consiga ser o mais segura possível.

Atualmente através do *WebAssembly* é possível portar grande parte das aplicações diretamente através de compiladores como o *Emscripten* e executadas na *WEB* diretamente sem ser preciso grandes alterações.

Terminando, apesar de estar em desenvolvimento o *WebAssembly* promete unificar performance, compatibilidade e segurança em uma única linguagem através de sistemas como o *WASI* que permitem a interoperabilidade dos sistemas operativos. O que leva a futuras promessas de não só resolver essas limitações, mas também permitir que outras tecnologias fiquem disponíveis para uso, introduzindo assim a próxima secção, trabalhos futuros para esta linguagem.

6.2 Trabalho Futuro

Infelizmente, pelo facto de ainda não existir acesso a algumas tecnologias por parte do *WebAssembly*, não se fez alguns testes de performance por um impasse que originou limitações, como aconteceu no teste de *RSA* onde não houve nenhum exemplo em *Rust* por *bugs* na biblioteca usada. Derivado a este problema espera-se que no futuro sejam possíveis novos testes neste campo, como por exemplo o uso do *Openssl* através do *Rust* para *WebAssembly*.

Por outro lado, evitando impassses como o anterior, seria interessante ser feita uma análise a algumas *crypto-miners* em *WebAssembly* através de ferramentas como o *WASABI* ou o *Manticore*, como é referido no final do capítulo quatro (4.3.4).

Dentro do tópico do *WebAssembly* e no âmbito de testar o comportamento das ferramentas desenvolvidas, seria interessante fazer uma comparação entre vários *standalones* existentes, especialmente entre o *Wasmer*[10] e o *Wasmtime* como podemos ver apresentado neste artigo[27].

Para quando for possível ter um sistema de *networking* completamente implementado em *WebAssembly*, seria implementar um sistema de mensagens por texto ou um *chat* que através de ferramentas como o *WASI* poderiam ser criados de forma uniforme desenvolvendo o código para a plataforma *WASI* e esta seria encarregada de traduzir para o sistema dispositivo que estaria a usa-la, poupando assim o trabalho do programador ter que desenvolver aplicações diferentes para sistemas diferentes, visto que, essa última parte sê-lo-ia entregue ao *WASI*.

Bibliografia

[1] Pedro Lopes. Projeto-WASM, 2021. [Online] https://github.com/arctumn/Projeto-WASM. Último acesso a 25 de janeiro de 2021.

[2] The Rust Project Developers. The Rustonomicon. [Online] https://doc.rust-lang.org/nomicon/leaking.html. Último acesso a 10 de novembro de 2020.

[3] Google. V8 - Google's open source high-performance JavaScript and WebAssembly engine. [Online] https://v8.dev/. Último acesso a 2 de fevereiro de 2021.

[4] LLVM Admin Team. LLVM. [Online] https://llvm.org/. Último acesso a 11 de novembro de 2020.

[5] Emscripten Contributors. About Emscripten. [Online] https://emscripten.org/docs/introducing_emscripten/about_emscripten.html. Último acesso a 11 de novembro de 2020.

[6] bytecodealliance. WebAssembly System Interface. [Online] https://wasi.dev/. Último acesso a 2 de fevereiro de 2021.

[7] bytecodealliance. Wasmtime - A small and efficient runtime for WebAssembly and WASI. [Online] https://wasmtime.dev/. Último acesso a 2 de fevereiro de 2021.

[8] LLVM Developer Group. Clang: a C language family frontend for LLVM. [Online] https://clang.llvm.org/. Último acesso a 2 de fevereiro de 2021.

[9] bytecodealliance. WASI-SDK. [Online] https://github.com/WebAssembly/wasi-sdk. Último acesso a 2 de fevereiro de 2021.

[10] Wasmerio. Wasmer. [Online] https://wasmer.io/. Último acesso a 10 de fevereiro de 2021.

[11] WABT Dev Team. Matplotlib. [Online] https://matplotlib.org/. Último acesso a 25 de janeiro de 2021.

[12] WebAssembly Dev Team. Wasm text format. [Online] https://webassembly.github.io/spec/core/text/#. Último acesso a 25 de janeiro de 2021.

[13] WebAssembly Dev Team. Wasm binary format. [Online] https://webassembly.github.io/spec/core/binary/#. Último acesso a 25 de janeiro de 2021.

[14] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960.

[15] Valgrind™ Developers. Valgrind. [Online] https://valgrind.org/. Último acesso a 11 de novembro de 2020.

[16] Daniel Lehmann and Michael Pradel. Wasabi:A Framework for Dynamically Analyzing WebAssembly, 2019. [Online] http://software-lab.org/publications/asplos2019_Wasabi.pdf. Último acesso a 13 de novembro de 2020.

[17] SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks. [Online] https://personal.utdallas.edu/~hamlen/wang18esorics.pdf. Último acesso a 25 de janeiro de 2021.

[18] trailofbits. Manticore. [Online] https://github.com/trailofbits/manticore. Último acesso a 10 de fevereiro de 2021.

[19] rustwasm. Twiggy. [Online] https://github.com/WebAssembly/wasi-sdk. Último acesso a 10 de fevereiro de 2021.

[20] Abhinav Jangda, Bobby Powers, Arjun Guha, and Emery Berger. Mind the gap: Analyzing the performance of webassembly vs. native code. *CoRR*, abs/1901.09056, 2019.

[21] Peter Astrand. subprocess - Subprocesses with accessible I/O streams. [Online] https://docs.python.org/3/library/asyncio-subprocess.html. Último acesso a 25 de janeiro de 2021.

[22] et al Michael Droettboom. Matplotlib. [Online] https://matplotlib.org/. Último acesso a 25 de janeiro de 2021.

[23] tiny-AES-c Developers. tiny-AES-c, 2019. [Online] https://github.com/kokke/tiny-AES-c. Último acesso a 1 de dezembro de 2020.

[24] RustCrypto Developers. RustCrypto: Advanced Encryption Standard (AES), 2020. [Online] https://github.com/RustCrypto/block-ciphers/tree/master/aes. Último acesso a 1 de dezembro de 2020.

[25] Navin Maheshwari. Rsa 1024bit in C for Embedded Systems. [Online] https://github.com/navin13692/RSA-1024bit-in-C-. Último acesso a 25 de janeiro de 2021.

[26] WebAssembly Community Group. WebAssembly Garbage Collector. [Online] https://github.com/WebAssembly/gc/blob/master/proposals/gc/MVP.md. Último acesso a 10 de fevereiro de 2021.

[27] SimonHeath. ActuallyUsingWasm. [Online] https://wiki.alopex.li/ActuallyUsingWasm. Último acesso a 10 de fevereiro de 2021.