Departamento de Informática

WEBASSEMBLY

WebAssembly

Elaborado por:

Pedro Manuel Pires Lopes

Orientador:

Professor Doutor Paul Crocker

Introdução

Este projeto tem como objetivo apresentar a linguagem *WebAssembly* e um conjunto de ferramentas existentes que se encontram a ser desenvolvidas para esta linguagem.

- WebAssembly – Os Básicos.
  - O que é a linguagem *WebAssembly*;
  - O ecossistema WebAssembly :
    - Compilar, executar e inspecionar *WebAssembly*;
- Análise dinâmica
  - Analise dinâmico de código usando a ferramenta *WASABI;*
- Estudo da performance
  - Desempenho entre o código nativo, o código gerado para *WebAssembly* e o *JavaScript;*
- Resumo e Conclusões
  - Limitações atuais e o Futuro do *WebAssembly*.

FACULDADE ENGENHARIA
Departamento de Informática

O que é o *WebAssembly*?

- **É uma linguagem de programação rápida, eficiente e portável**
- **É uma linguagem para complementar a *Web* com o *JavaScript***
- **É uma linguagem retro compatível**
- **É uma linguagem que dá suporte a outras linguagens na *Web***

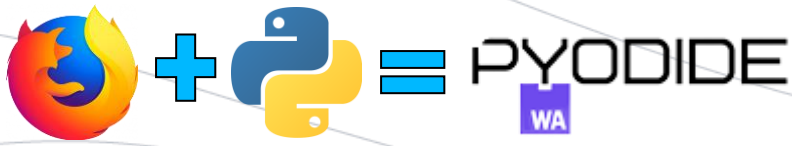
FACULDADE ENGENHARIA
Departamento de Informática

Compilar *WebAssembly* para a *Web*

Segundo o website do *WebAssembly* estes são algumas das linguagens, apesar de já existirem mais.

- C/C++
  - starting from scratch
  - library that I want to port to the Web
- Rust
- AssemblyScript (a TypeScript-like syntax)
- C#
- F#
- Go
  - with full language support
  - targeting minimal size
- Kotlin
- Swift
- D
- Pascal
- Zig

Por exemplo *através do* Pyodide permite executar *Python* na *Web* (compilando o *Cpython* para *WebAssembly pelo Emscripten*).



FACULDADE ENGENHARIA
Departamento de Informática

Compilar *WebAssembly* para a *Web*

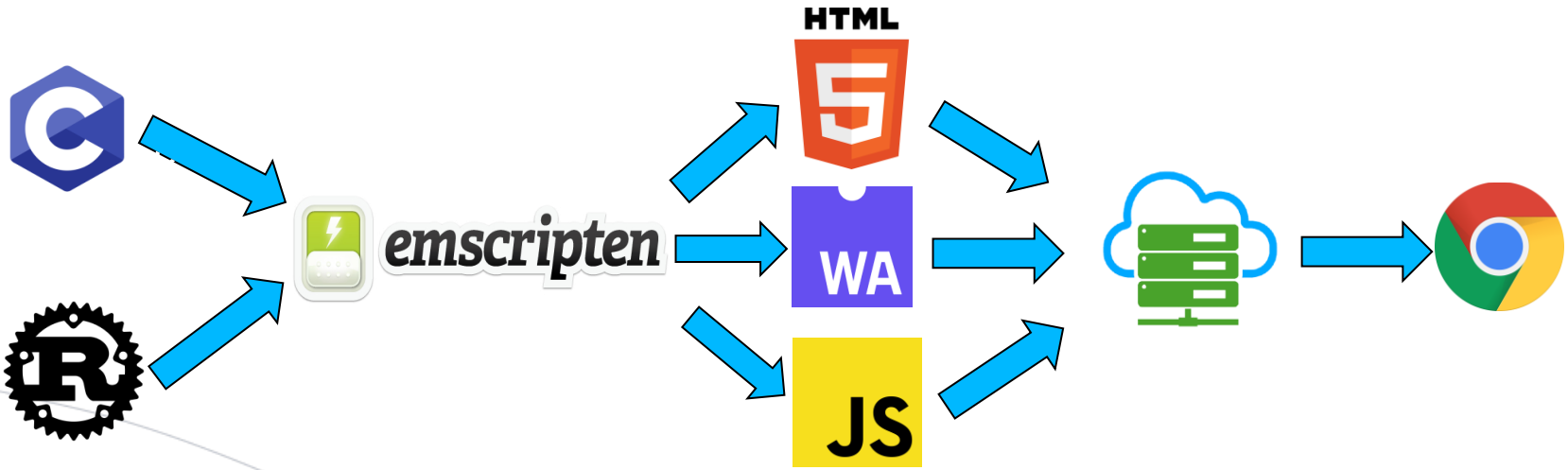
Apesar do grande suporte do *WebAssembly para várias linguagens, o maior foco de desenvolvimento encontra-se nas linguagens* C *e Rust*.


Derivado desta situação temos o *Emscripten* como um dos métodos mais escolhidos para compilar para *WebAssembly derivado a sua simplicidade de compilar e executar sem grandes dificuldades*.

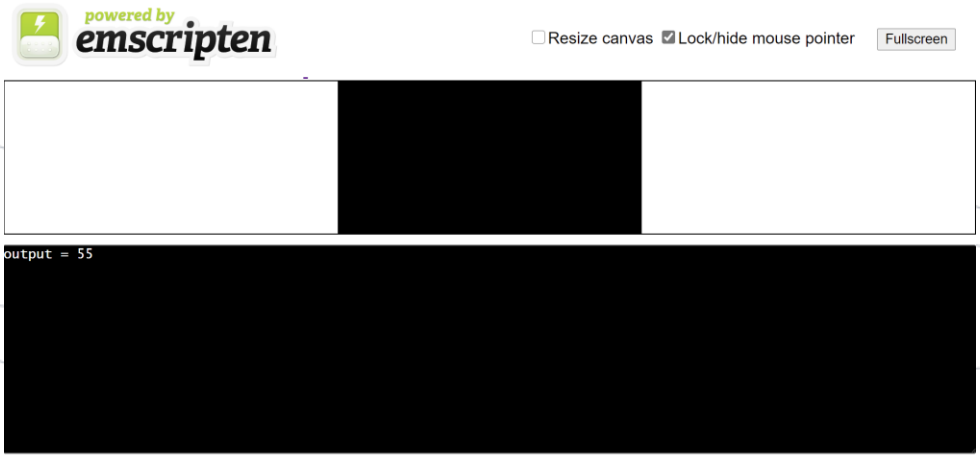
FACULDADE ENGENHARIA
Departamento de Informática

Compilar *WebAssembly* para a *Web*

O diapositivo anterior pode ser resumido pelo seguinte esquema:



Originando um modelo de página como a próxima figura:



FACULDADE ENGENHARIA
Departamento de Informática

Compilar *WebAssembly* para a *Web*

Obviamente que é possível incorporar o *WebAssembly* como uma *library* onde o *JavaScript acede as funções compiladas*.

Através de esquemas como acontece na figura abaixo:

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

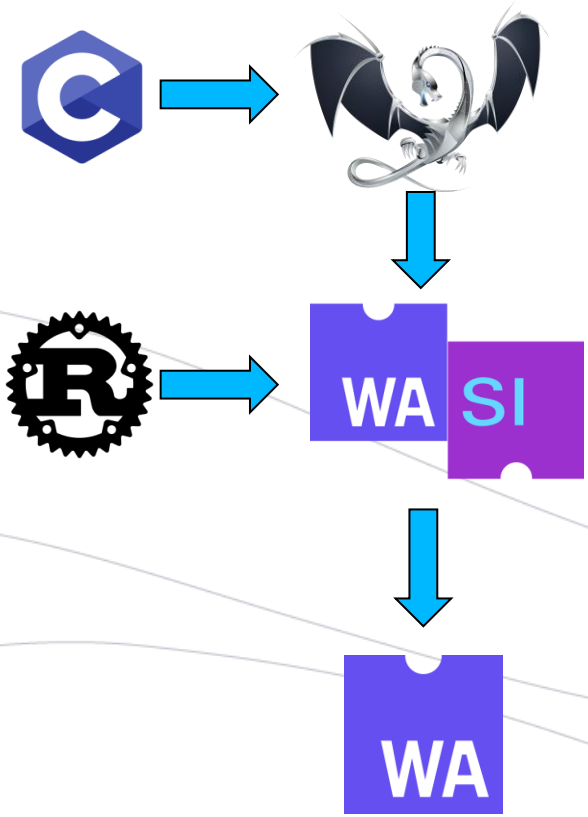
Desta forma o *JavaScript* acede ao modulo, e o programador pode incorporar com os seus *scripts, obtendo uma performance superior em relação a* JavaScript *nativo*.

FACULDADE ENGENHARIA
Departamento de Informática

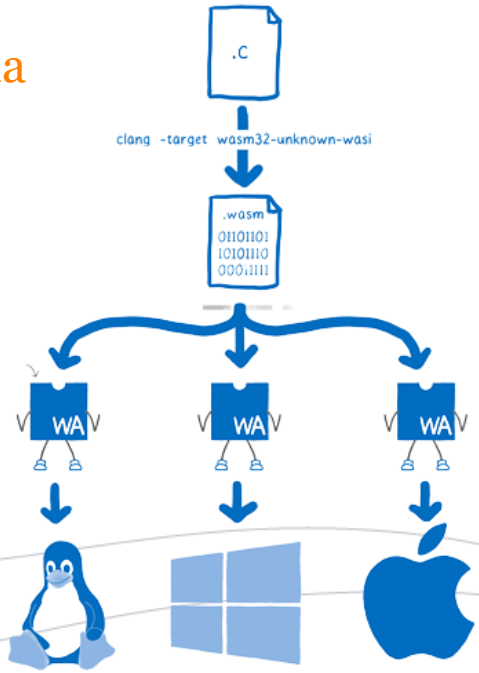
Compilar *WebAssembly* fora da *Web*

Podemos usar o ***WASI*** através de duas ferramentas:

- ***Clang***
- ***Rust-C***



Através do esquema feito pela Mozilla obtemos a formula exata do funcionamento do *WASI*.



FACULDADE ENGENHARIA
Departamento de Informática

Executar WebAssembly fora da *Web*

Para responder ao *WASI* foram criadas algumas ferramentas das quais se destacam duas que derivam da mesma origem, mas com responsáveis diferentes, estas são:

- *Wasmtime*

- *Wasmer*

Ambos os *runtimes* têm capacidade de executar *WebAssembly*.

O Wasmtime é desenvolvido pelo mesmo grupo do *WASI*, o *Wasmer* é desenvolvido por um grupo independente.

FACULDADE ENGENHARIA
Departamento de Informática

Executar WebAssembly fora da *Web*

Tanto o *Wasmtime* como o *Wasmer* cumprem as funcionalidades fornecidas pelo *WASI*.

```
arctumn@LAPTOP-1QNTV8EU:~/Projeto-WASM/wasi/wasi-c$ wasmtime teste.wasm
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

arctumn@LAPTOP-1QNTV8EU:~/Projeto-WASM/wasi/wasi-c$ wasmtime --dir=. teste.wasm
arctumn@LAPTOP-1QNTV8EU:~/Projeto-WASM/wasi/wasi-c$
```

Para aceder ao sistema de ficheiros ambos usam a *flag* “*–dir=*“ sem ela não deixam o *WebAssembly* aceder aos ficheiros.

```
arctumn@LAPTOP-1QNTV8EU:~/Projeto-WASM/wasi/wasi-c$ wasmer teste.wasm
error: failed to run `teste.wasm`
    1: WASI execution failed
    2: failed to run WASI `_start` function
    3: RuntimeError: uninitialized element
           at fclose (teste.wasm[35]:0x374b)
           at factorial (teste.wasm[15]:0x4e0)
           at main (teste.wasm[16]:0x555)
           at __main_void (teste.wasm[26]:0x34f0)
           at __original_main (teste.wasm[17]:0x574)
           at _start (teste.wasm[13]:0x2cd)
└─> 4: icall_null
arctumn@LAPTOP-1QNTV8EU:~/Projeto-WASM/wasi/wasi-c$ wasmer --dir=. teste.wasm
arctumn@LAPTOP-1QNTV8EU:~/Projeto-WASM/wasi/wasi-c$
```

FACULDADE ENGENHARIA
Departamento de Informática

Executar WebAssembly fora da *Web*

Para além de executarem *WebAssembly*, estes têm mais algumas funcionalidades como inspecionar o código do ficheiro *WebAssembly*.

O *Wasmtime* tem a capacidade de produzir ficheiros em formato *object* que podem ser analisados através do *objdump em Assembly*.

Por outro lado o *Wasmer* tem a capacidade de extrair a informação, mostrando a assinatura de funções, memória e até variáveis globais.

FACULDADE ENGENHARIA
Departamento de Informática

Executar WebAssembly fora da *Web*

Como referido no diapositivo anterior, podemos ver a informação de algumas funções executadas para *WebAssembly* pelo *Wasmer*.

Standalone pelo *Emscripten* + WASI

```
arctumn@LAPTOP-1QNTV8EU:~/Projeto-WASM/wasi/wasi-t$ wasmer inspect emcc-teste.w
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)
```

Standalone pelo *Clang* + *WASI*

```
arctumn@LAPTOP-1QNTV8EU:~/Projeto-WASM/wasi/wasi-t$ 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:
```

FACULDADE ENGENHARIA
Departamento de Informática

Inspecionar WebAssembly

Para poder-mos ter outra perspetiva de inspecionar temos a ferramenta *WABT (The WebAssembly Binary Toolkit).*

Esta ferramenta permite em converter o *WebAssembly* que se encontra no formato binário e converter para o formato de texto e vice-versa.

O *WABT* em formato de texto aparece no formato de *S-expression*, através deste mesmo formato pode-se produzir WebAssembly.

WebAssembly escrito em wat

```
arctumn@LAPTOP-1QNTV8EU:~/pp/Projeto-WASM/wabt$ cat teste.wat
(module
  (func $mulby2 (param f32) (result f32) ;; nome_funcao param_in param_out
    local.get 0 ;; param num 0
    f32.const 2
    f32.mul ;; multiplica os elementos atras mul apenas aceita dois argumentos
  )
  (func (export "sum") (result f32) ;; export para JS param_out
    f32.const 5
    call $mulby2 ;; multiplica 5 por 2
  )
)
```

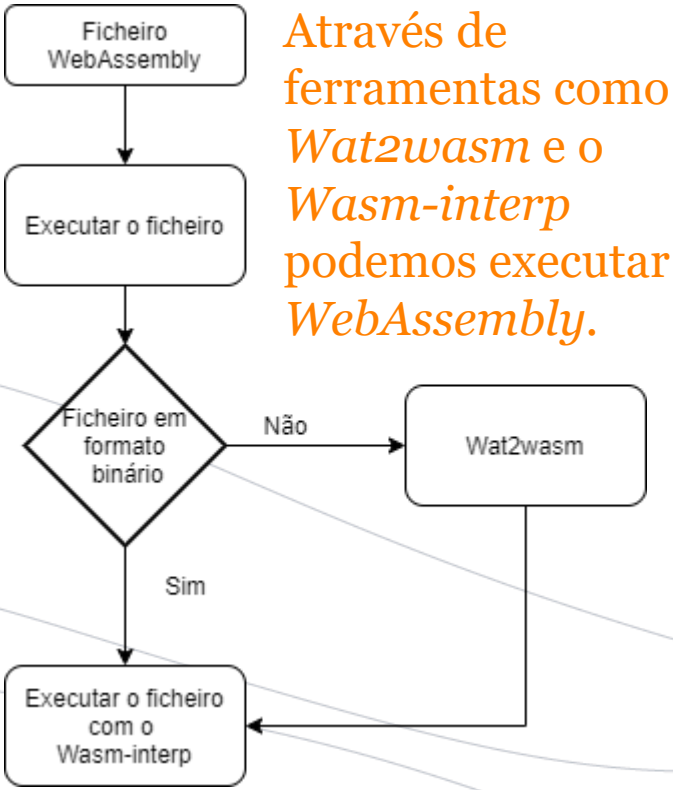
WebAssembly convertido para wat

```
arctumn@LAPTOP-1QNTV8EU:~/pp/Projeto-WASM/wabt$ cat teste2.wat
(module
  (type (;0;) (func (param f32) (result f32)))
  (type (;1;) (func (result f32)))
  (func (;0;) (type 0) (param f32) (result f32)
    local.get 0
    f32.const 0x1p+1 (;=2;)
    f32.mul)
  (func (;1;) (type 1) (result f32)
    f32.const 0x1.4p+2 (;=5;)
    call 0)
  (export "sum" (func 1)))
```

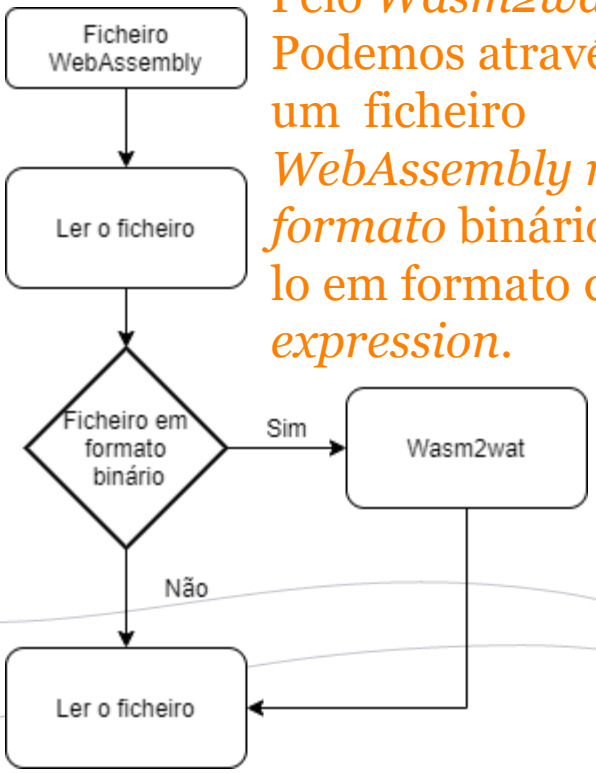
FACULDADE ENGENHARIA
Departamento de Informática

Inspecionar WebAssembly

Como referido na diapositivo anterior, o *WABT* consegue converter o *WebAssembly* em dois formatos existentes.



Através de ferramentas como o *Wat2wasm* e o *Wasm-interp* podemos executar *WebAssembly*.



Pelo *Wasm2wat* Podemos através de um ficheiro *WebAssembly no formato* binário e lê-lo em formato de *S-expression*.

Análise dinâmica na ferramenta *WASABI*

Para análises dinâmicas, ferramentas como o *Wasmer* ou o *WABT* não oferecem uma interface propriamente simples.

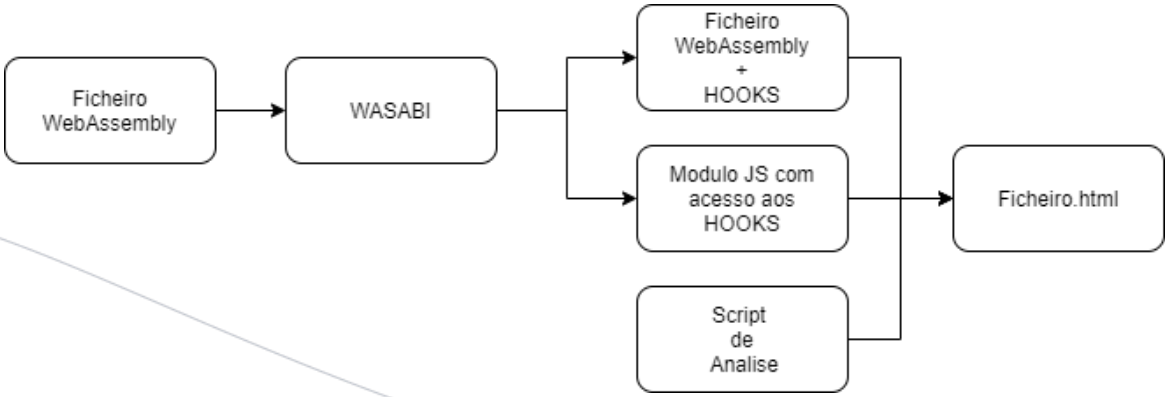
- *WASABI*
- *Manticore*
- *Twiggy*

Infelizmente o *Twiggy* apenas trabalha em *Rust* antes do código ser compilado.

FACULDADE ENGENHARIA
Departamento de Informática

Análise dinâmica na ferramenta *WASABI*

O WASABI(WebAssembly analysis using binary instrumentation) permite aos seus utilizadores fazerem análises específicas em *JavaScript* para *WebAssembly*.



Existe um artigo atualmente que utiliza a ferramenta *WASABI*, para analisar a quantidade de instruções de diversas *cripto-miners* que usam *WebAssembly*.

Análise dinâmica na ferramenta *WASABI*

Neste caso iremos criar um *script* que pega na execução do *WebAssembly* e extrai assinatura das funções que são executadas.

O alvo de teste foi feito em *C* e consiste uma função que calcula o 10 elemento da sequência de *fibonnaci* recursivamente.

```c
#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;
}
```

FACULDADE ENGENHARIA
Departamento de Informática

Análise dinâmica na ferramenta *WASABI*

Para fazer a análise iremos usar as seguintes funções em *JavaScript:*

- `function fctName(fctId)`
- `function parseType(serializedType)`
- `function argType(fctId)`

A `function fctName(fctId)` como o nome indica tenta extrair o nome da função que vai ser usada.

```
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;
    return fctId;
}
```

Esta pega nas informações obtidas pelo *WASABI* e retorna o nome delas.

Pelo módulo do WASABI esta é chamada da seguinte forma:

```
Wasabi.analysis = {
    call_pre(location, targetFunc, _notused, _notused2) {
        const caller = fctName(location.func);
        const callee = fctName(targetFunc);
        const argcallee = argType(targetFunc);
        listOfElements.push(
            "The function "
            + caller
            + " is using the function "
            + callee
            + "("+argcallee+")"
        );
    },
};
```

FACULDADE ENGENHARIA
Departamento de Informática

Análise dinâmica na ferramenta *WASABI*

A função `function parseType(serializedType)` pega no tipo criado pelo *WASABI*, onde extrai e organiza os tipos das funções e os seus 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':
                    parsedString = parsedString + "i32 "
                    break;
                case 'I':
                    parsedString = parsedString + "i64 "
                    break;
                case 'f':
                    parsedString = parsedString + "f32 "
                    break;
                case 'F':
                    parsedString = parsedString + "f64 "
                    break;
                case '|':
                    parsedString = parsedString + "; Returning args: "
                    break;
                default:
                    break;
            }
        })

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

    return parsedString
    }
```

Esta função é chamada pela função `function argType(fctId)` que automatiza o envio dos tipos como podemos ver abaixo.

```
function argType(fctId){
    const fct = Wasabi.module.info.functions[fctId];
    return parseType(fct.type)
}
```

Análise dinâmica na ferramenta *WASABI*

Com estas três funções conseguimos observar o comportamento das funções no *WebAssembly*.

Através do *script* podemos obter a informação no seguinte ficheiro:

```
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 )
```

Do ficheiro resultante observa-se:

- Recursão da função *fibonnaci (7)*
- Chamada da função (7) pelo *printf (8)*
- Chamada do *printf* pela *main*

Análise dinâmica na ferramenta *WASABI*

Adaptando a função *fctName*, é possível fazer *scripts* que contam as chamadas ao *malloc*, ao *free* e a outros, permitindo ao programador detetar falhas na memória presentes no *WebAssembly*.

Por fim, o *WASABI* permite de forma rápida criar *scripts* que recolhem a informação e a analisem o *WebAssembly* de forma detalhada sem muita dificuldade.

FACULDADE ENGENHARIA
Departamento de Informática

Performance do *WebAssembly*

Com o *WebAssembly* podemos otimizar o *JavaScript* permitindo a execução de tarefas de forma mais eficiente.

Na realidade é preciso avaliar a perda de performance em relação ao código nativo.

Num estudo realizado sobre performance de *WebAssembly* em matrizes encontrou-se uma perda de aproximadamente 50% de performance em relação ao mesmo código em *C*.

Performance do *WebAssembly*

Vão ser apresentados os resultados dos testes na seguinte ordem, primeiro em *AES-128* e depois em *RSA-1024*:

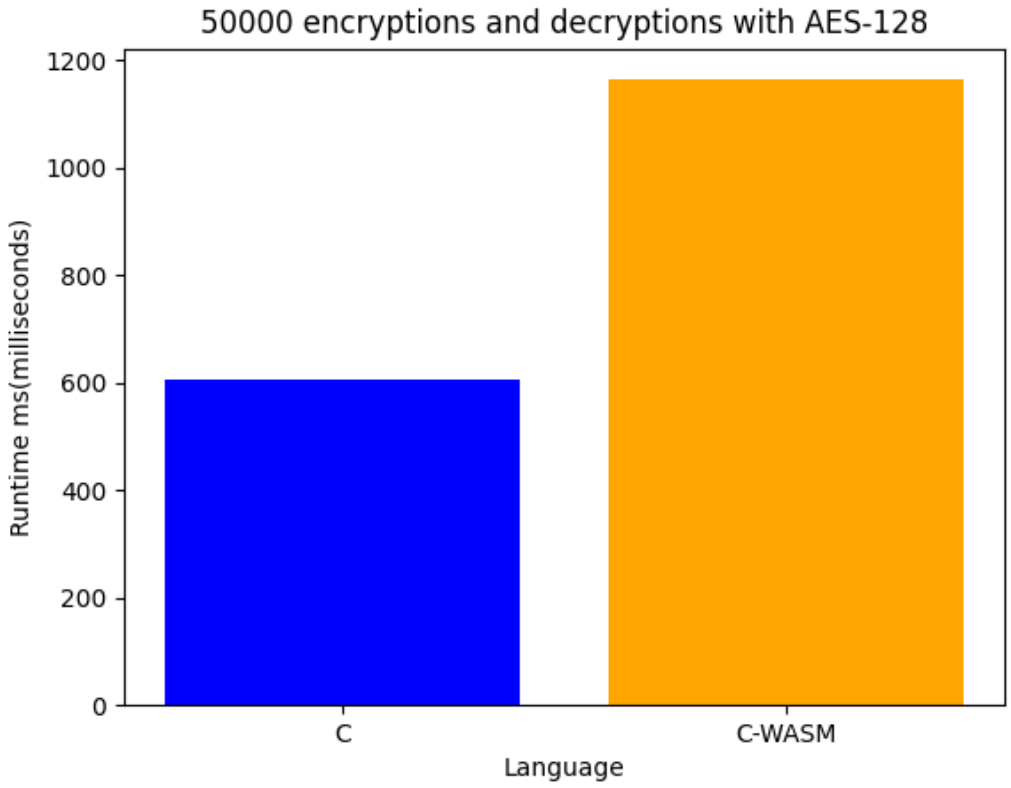
- *AES-128 C* e *C-WASM*
- AES-128 Rust e Rust-WASM
- *AES-128 C-WASM*, *Rust-WASM*, *JavaScript*
- *RSA-1024 C* e *C-WASM*
- *RSA-1024 C-WASM* e *JavaScript*

Não foi introduzido um teste em *Rust* para *RSA-1024* derivado a um problema com o *OpenSSL*.

FACULDADE ENGENHARIA
Departamento de Informática

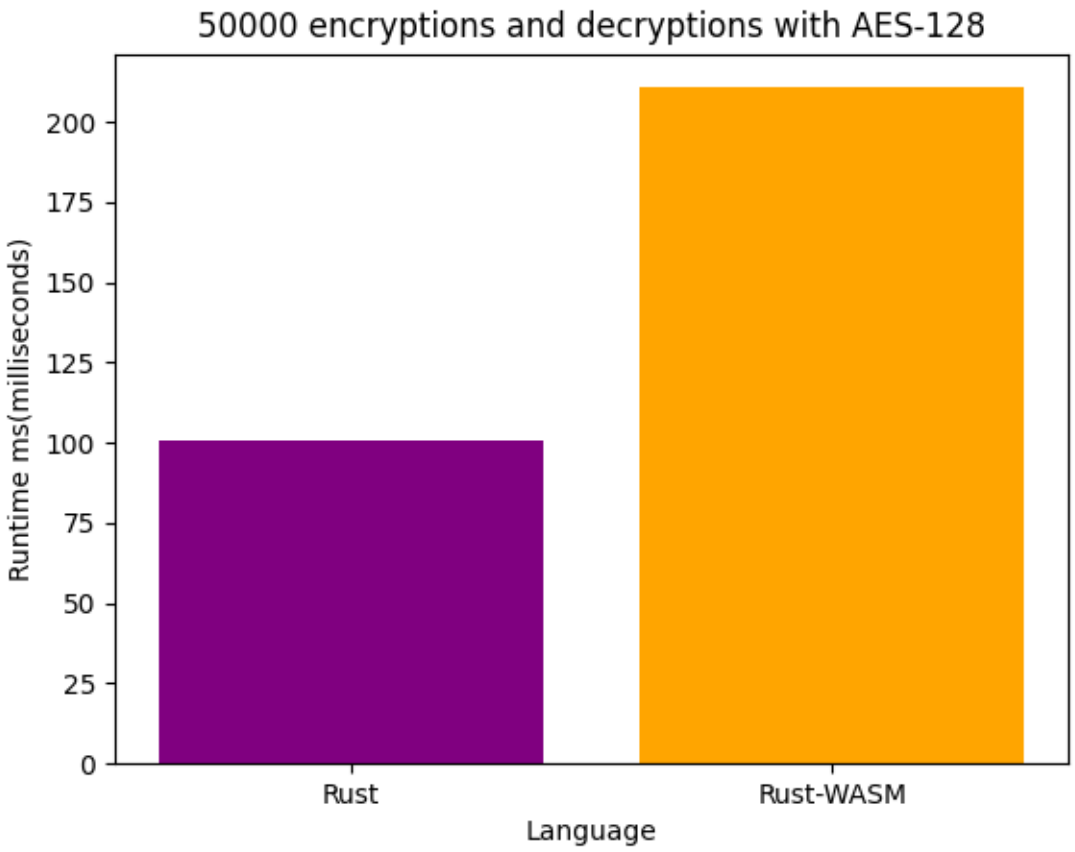
Performance do *WebAssembly*

AES-128 C e C-WASM



Performance do *WebAssembly*

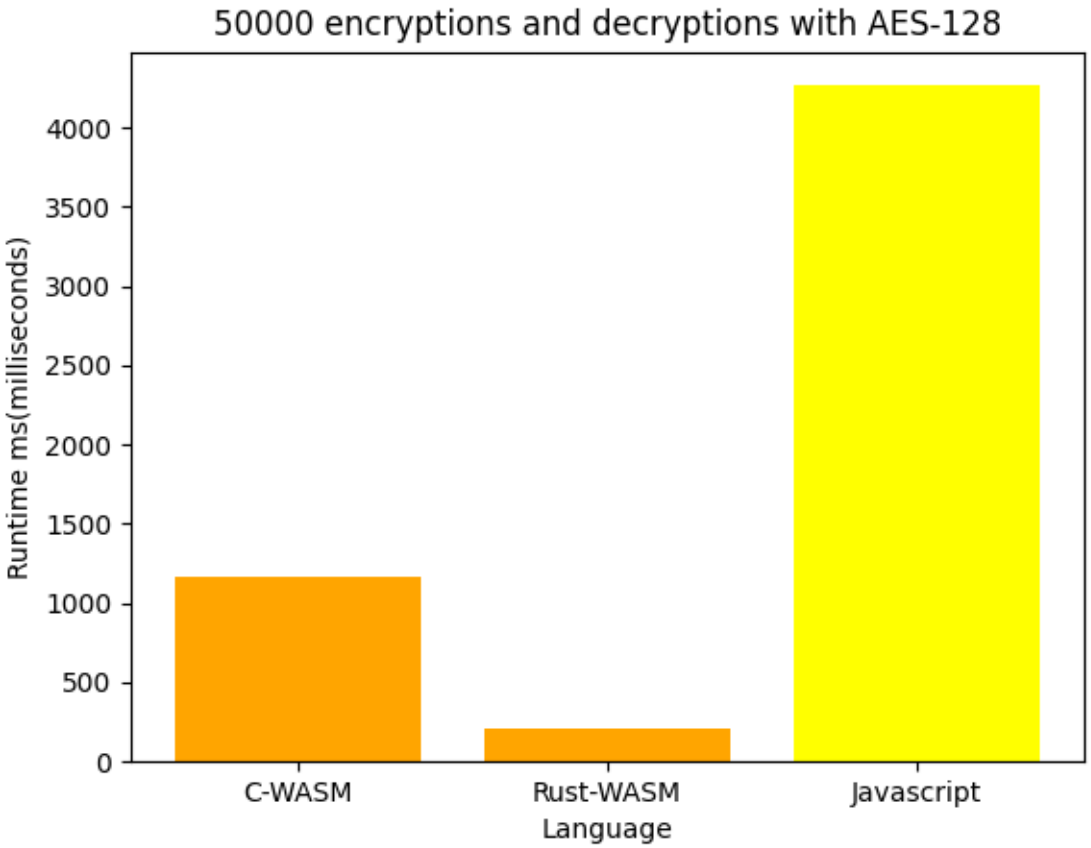
AES-128 Rust e Rust-WASM



FACULDADE ENGENHARIA
Departamento de Informática

Performance do *WebAssembly*

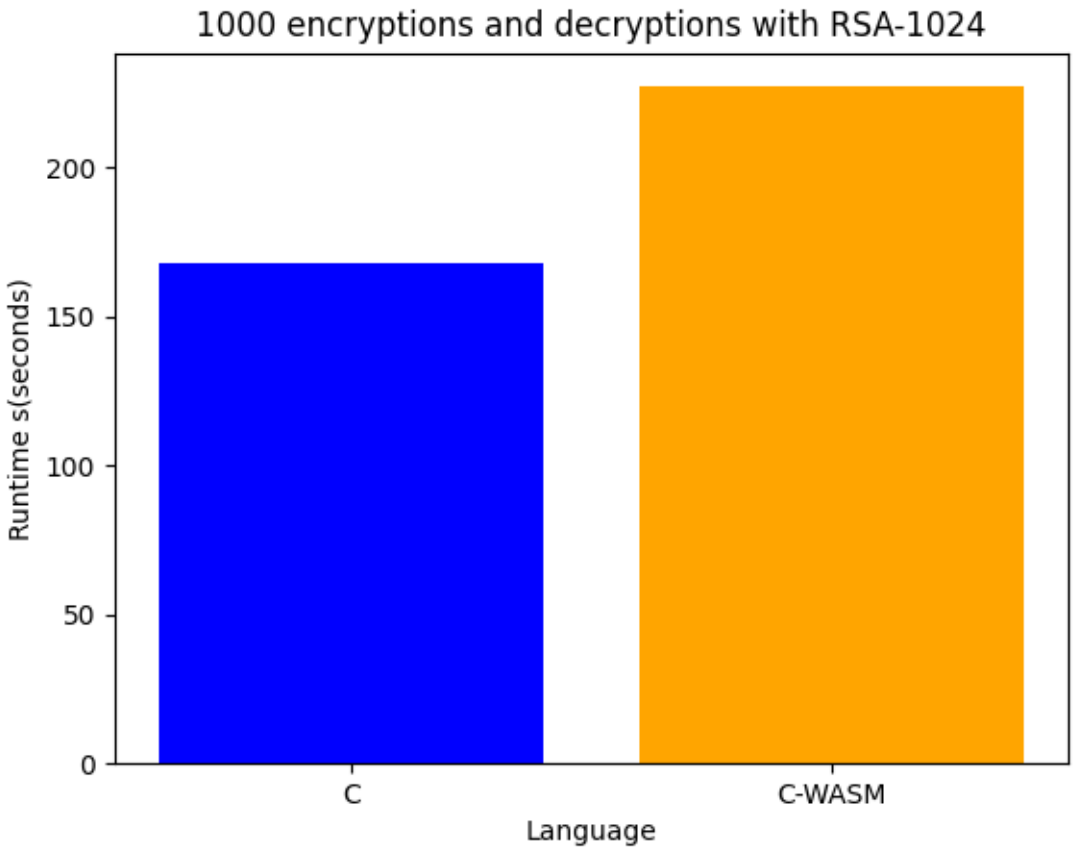
AES-128 C-WASM, Rust-WASM e Javascript



FACULDADE ENGENHARIA
Departamento de Informática

Performance do *WebAssembly*

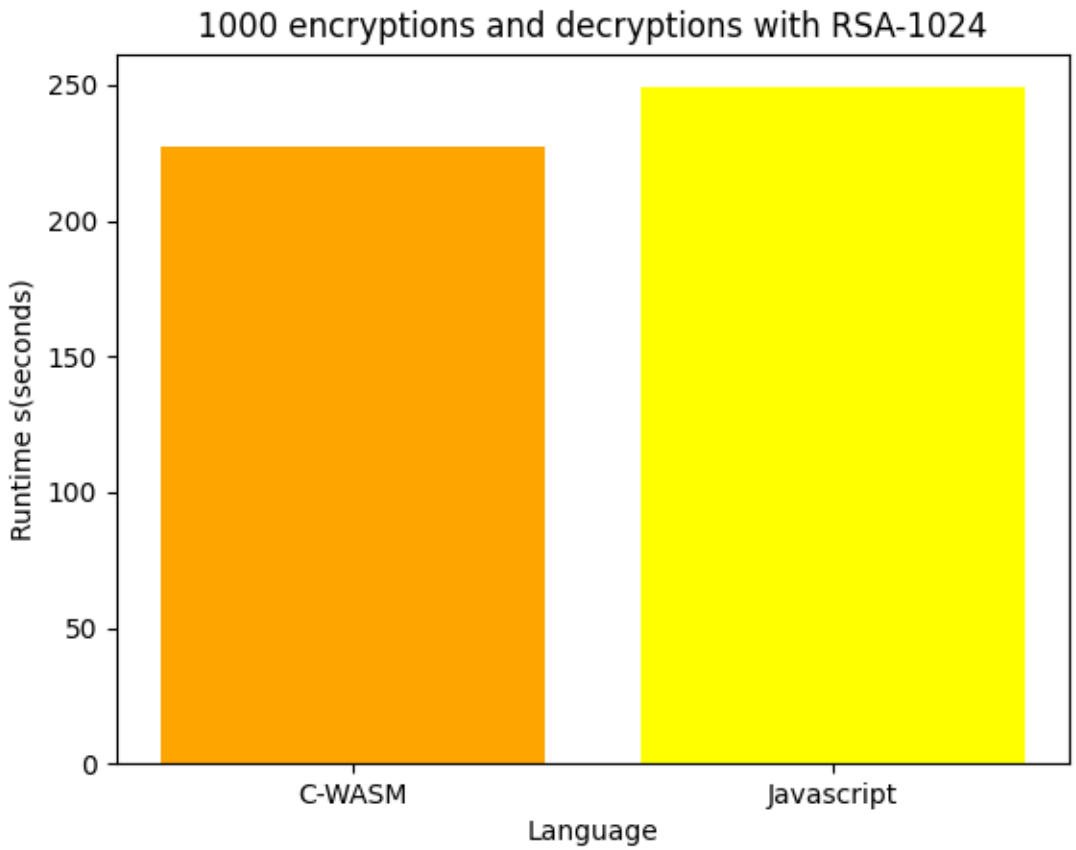
RSA-1024 C e C-WASM



FACULDADE ENGENHARIA
Departamento de Informática

Performance do *WebAssembly*

RSA-1024 C-WASM e JAVASCRIPT



FACULDADE ENGENHARIA
Departamento de Informática

Performance do *WebAssembly*

AES-128* e *RSA-1024

- Impacto de performance entre 35%-65% mais lento que o nativo.
- Programas em *WebAssembly* são mais rápidos.
- *WebAssembly* permite ao JavaScript margem de otimizações as suas funções.
- Possível oportunidade para o *WebAssembly* aproximar-se do código nativo.

Limitações e Futuro do *WebAssembly*

Atualmente existem ainda algumas limitações em via de serem resolvidas na linguagem *WebAssembly:*

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