Capítulo 6

WebAssembly: Uma Introdução

Tiago Heinrich[1], Beatriz M. Reichert[2], Newton C. Will[3], Rafael R. Obelheiro[2], Carlos A. Maziero[4]

[1]Max Planck Institute for Informatics (MPI)
[2]Universidade do Estado de Santa Catarina (UDESC)
[3]Universidade Tecnológica Federal do Paraná (UTFPR)
[4]Universidade Federal do Paraná (UFPR)

Abstract

In the last decade, Web browsers have become an indispensable resource for users to access the Internet and carry out everyday activities. During this period, different resources were proposed to gain performance, security and practicality in the development of Web applications. One of these resources is WebAssembly, which is a binary and portable format. In this chapter, a complete contextualization of the history of the Web up to its current state will be introduced, with a focus on introducing the WebAssembly format and the impact that this new feature brings to the Web. Discussing performance factors, security, recent trends and open issues.

Resumo

Na última década os navegadores Web se tornaram um recurso indispensável para usuários acessarem a Internet e realizarem atividades cotidianas. Ao decorrer deste período diferentes recursos foram propostos para ganho de desempenho, segurança e praticidade no desenvolvimento de aplicações Web. Um destes recursos é o WebAssembly*, que é um formato binário e portátil. Neste capítulo, será introduzido toda uma contextualização da história da Web até o estado atual, com o foco em introduzir o formato* WebAssembly *e impacto que este novo recurso traz para a Web. Discutindo sobre fatores de performance, segurança, tendências recentes e problemas abertos.*

6.1. Introdução

Inicialmente, a Web era formada por conteúdo exclusivamente estático: páginas HTML com *hiperlinks* e imagens que eram exibidas em um navegador, e arquivos binários que podiam ser salvos para uso local [Fox and Patterson 2021]. Com o decorrer dos anos, as páginas Web passaram a ser dependentes de conteúdos dinâmicos, que são gerados no momento do acesso, muitas vezes de forma personalizada para o usuário. Esses conteúdos dinâmicos podem ser gerados do lado do servidor ou do cliente, mediante a execução de código no contexto do navegador [Stock et al. 2017]. Atualmente, o conteúdo disponível para os usuários ao navegarem pela Web tem o predomínio de conteúdo dinâmico nas páginas que requerem a execução de operações no lado do usuário.

Devido a este "novo" ecossistema, novas linguagens foram introduzidas com o passar do tempo. Historicamente, três linguagens foram propostas com foco no desenvolvimento Web do lado do cliente, sendo elas: Java applets, Flash e JavaScript (JS) [Golsch 2019]. Devido à utilização de diferentes estratégias/*designs*, cada linguagem possui suas peculiaridades, limitações e políticas de segurança.

Java applets e Flash foram descontinuados para aplicativos da Web devido a problemas de desempenho e segurança, respectivamente. JS ganhou então exclusividade em navegadores Web [Feldman 2019]. Dados de dezembro de 2023 estimam que 98,8% dos websites usam JS no lado do cliente [W3Techs 2023].

JS é uma linguagem de fácil uso, que permite o desenvolvimento de aplicativos de pequeno e médio porte. A eficiência e a segurança da linguagem constituem desafios importantes, particularmente em aplicações complexas. Dois problemas encontrados em JS são a falta de tipos e a sobrecarga do interpretador de tempo de execução decorrente da falta de um formato binário. Esses problemas foram identificados em 2015, levando grandes organizações desenvolvedoras a propor o WebAssembly como uma alternativa para superar as limitações encontradas no JS [Bandhakavi et al. 2010, Fraiwan et al. 2012, Yan et al. 2021].

WebAssembly foi lançado em 2017, visando propor um ambiente com maior segurança, velocidade e semântica portátil [Golsch 2019]. Trata-se de um formato de código binário portátil (*bytecode*) projetado para execução segura e eficiente, com uma representação compacta [Rossberg 2024]. Normalmente, ele é usado como um formato de código executável gerado por compiladores para linguagens de alto nível, como C, C++, Go e Rust. Dessa forma, o WebAssembly permite que aplicações e bibliotecas desenvolvidas nessas linguagens sejam executadas em navegadores Web ou mesmo em ambientes nativos [Hoffman 2019].

O *bytecode* WebAssembly geralmente é executado dentro de uma *sandbox* chamada WebAssembly *runtime*. Essa escolha de *design* permite que os aplicativos sejam executados em uma ampla variedade de plataformas, reduz o tamanho das mensagens a serem trocadas com o servidor para recuperar conteúdos, e fornece maior segurança para o ambiente do cliente que executa o conteúdo.

Atualmente, o formato WebAssembly já é suportado por todos os maiores navegadores, como Mozilla Firefox, Google Chrome e Safari [W3C Community Group 2022b]. A gama de aplicações que usam WebAssembly inclui criptografia, processamento de ví-

deo/áudio/gráficos e outras atividades que podem aproveitar as vantagens do *design* do formato. No entanto, o formato não se limita aos navegadores Web. Aplicativos do lado do servidor, serviços distribuídos, Internet of Things (IoT), Cloud e aplicativos móveis também podem aproveitar as vantagens do WebAssembly [Rossberg 2024]. Considerando a rápida adoção do WebAssembly, é importante conhecer as características desta tecnologia e entender seus benefícios e limitações, particularmente no que tange a desempenho e segurança, que são geralmente apontados como suas principais virtudes.

O *design* do WebAssembly conta com diversos mecanismos para proporcionar mais segurança e desempenho para aplicações Web. No entanto, o WebAssembly também pode ser explorado como um vetor de ataques. Este minicurso faz uma introdução à tecnologia WebAssembly, focando nos seguintes objetivos:

Objetivos. Este curso é proposto na modalidade teórica, visando contextualizar o WebAssembly na atualidade, discutindo os principais aspectos do formato, qual o processo de desenvolvimento para o formato, contribuições já propostas para o formato WebAssembly, tendências recentes e problemas de pesquisa ainda em aberto.

Audiência. Este curso está destinado a qualquer público que esteja começando ou que tenha interesse em novas tecnologias encontradas no ecossistema da Web. Espera-se que, ao final do curso, os participantes tenham adquirido um conhecimento sobre WebAssembly que lhes permita compreender os objetivos e o papel desta tecnologia, distinguir seus principais benefícios e limitações, e embasar um maior aprofundamento no assunto.

Estrutura. O restante deste capítulo está organizado como segue. A Seção 6.2 faz um breve histórico do ecossistema Web. A Seção 6.3 discute as principais limitações encontradas no ambiente Web que contribuíram para o desenvolvimento do WebAssembly. O WebAssembly é apresentado na Seção 6.4. A Seção 6.5 aborda aspectos de desempenho do WebAssembly. A Seção 6.6 traz uma revisão abrangente de aspectos de segurança da tecnologia. A Seção 6.7 examina estratégias de mitigação, monitoramento e detecção de ameaças em WebAssembly. A Seção 6.8 discute tendências recentes e problemas abertos de pesquisa envolvendo WebAssembly. Por fim, a Seção 6.9 apresenta as considerações finais do capítulo.

6.2. História da Web

No início, a Web consistia em páginas de conteúdo estático, i.e., páginas HyperText Markup Language (HTML) com texto, imagens e *hiperlinks* que permitiam acesso a outras páginas ou arquivos (tipicamente arquivos binários que podiam ser salvos localmente pelo cliente, sem serem exibidos pelo navegador) [Fox and Patterson 2021]. A interação do usuário com as páginas resumia-se a acessar os *hiperlinks* disponíveis.

Com o decorrer dos anos, as páginas passaram a ser dependentes de conteúdos dinâmicos. A primeira fase consistia na execução de código do lado do servidor para gerar, *on-the-fly*, páginas HTML que implementavam uma interface com o usuário. O próximo passo na evolução foi o surgimento da tecnologia Asynchronous JavaScript and XML (AJAX). Com AJAX, as páginas passaram a incluir código JS que é executado no lado do cliente (no âmbito do navegador) para definir ou alterar o comportamento da página exibida [Fox and Patterson 2021]. Atualmente, observa-se que o conteúdo dinâmico

presente nas páginas é predominantemente gerado a partir de operações no lado do cliente [Stock et al. 2017].

Devido ao aumento na complexidade das aplicações Web, nota-se a exigência de um formato de entrega que seja portátil, compacto, rápido de executar e seguro. Tentativas anteriores de resolver estes problemas incluem JS, Java applets, Flash, ActiveX, Native Client (NaCl) e asm.js [Golsch 2019].

O JavaScript (JS) é a única linguagem de programação com suporte nativo na Web [Haas et al. 2017]. Atualmente é o principal componente das aplicações Web do lado do cliente. O código fonte é transmitido em texto simples e interpretado por um suporte de execução, o qual é executado isolado em uma *sandbox* e se comunica com o navegador da Web [Golsch 2019]. Entretanto, o JS é lento para a maioria das aplicações complexas, e se tornou um alvo de compilação para outras linguagens.

Java applets permitem a execução de código Java em navegadores da Web. Esse código é executado em uma Java Virtual Machine (JVM) isolada, a qual é integrada ao navegador ou pode ser instalada com um *plugin* [Golsch 2019]. Devido às vulnerabilidades de segurança, muitos navegadores não têm mais suporte aos Java applets.

O Flash possui uma abordagem semelhante ao JS, mas requeria a instalação de um *plugin* adicional para a execução [Golsch 2019]. O uso do Flash foi desaconselhado devido a problemas de segurança. Em dezembro de 2020 a Adobe anunciou o encerramento do suporte e da distribuição do Flash [Adobe 2021].

Em 1996, a Microsoft apresentou a abordagem ActiveX para assinatura de código de binários para execução na Web [Microsoft 1996]. Os objetos ActiveX incluem código de máquina compilado e, portanto, não são portáteis. Do ponto de vista de segurança, ActiveX é vulnerável pois a execução não é isolada do sistema operacional, como em uma *sandbox* [Golsch 2019].

Em 2009, a Google introduziu uma tecnologia chamada NaCl, que permite que código nativo seja executado em uma *sandbox* [Yee et al. 2010]. O código fonte pode ser compilado em um módulo, o qual é executado no ambiente interno do navegador Chromium. Como o NaCl é um subconjunto do código de máquina de uma arquitetura específica, ele não é portátil [Haas et al. 2017]. Posteriormente o Google introduziu o Portable Native Client (PNaCl), o qual é uma extensão do NaCl e permite a execução do código nativo independente de plataforma [Donovan et al. 2010]. Assim como o NaCl, o PNaCl é suportado apenas pelos navegadores Google Chrome e Chromium.

Em 2013, a Mozilla apresentou o asm.js [Herman et al. 2014], uma linguagem intermediária que, a partir do código nativo, gera sistematicamente um código usando um transcompilador (um compilador especializado em converter o código-fonte de um programa em outra linguagem fonte). Por exemplo, o Emscripten [Zakai 2011] pode ser usado para converter C/C++ em código JS e WebAssembly.

Após as experiências com Java, ActiveX, Flash, NaCl e asm.js, e considerando as limitações do JS, foi proposto o WebAssembly. Este consiste em um novo formato portátil, eficiente em tamanho e tempo de carregamento, adequado para compilação na Web. Em março de 2017 a Minimum Viable Product (MVP) foi concluída para o WebAssem-

bly, com o Safari já adicionando o suporte no final do mesmo ano. Em 2018 os primeiros *drafts* de especificações para a JS *Interface* e Web API foram apresentadas. O primeiro *draft* do WebAssembly 2.0 foi anunciado em 2022 (com a última atualização em março de 2024) [Rossberg 2024].

6.3. Limitações da Web

Os navegadores Web evoluíram significativamente nos últimos anos. Atualmente os navegadores são executados em diferentes tipos de *hardware*, como celulares, tablets e computadores [Grosskurth and Godfrey 2006]. No entanto, a maioria dos navegadores Web ainda usam uma arquitetura monolítica introduzida em 1993 pelo NCSA Mosaic [Barth et al. 2008, Rokicki 2022]. Do ponto de vista da segurança, os navegadores monolíticos são executados em um único domínio de proteção. Dessa forma, um invasor capaz de explorar uma vulnerabilidade não corrigida pode comprometer toda a instância do navegador. Além disso, um invasor pode executar código arbitrário na máquina do usuário com os privilégios deste.

Antes do WebAssembly houve outras tecnologias com foco no desenvolvimento Web do lado do cliente (*e.g.*, Flash, ActiveX, Java applets, asm.js, e JS). Entretanto, cada uma tem suas peculiaridades, limitações e políticas de segurança. O Flash, por exemplo, exigia que o usuário instalasse o Flash Player [Golsch 2019]. Além de apresentar carregamento lento e gerar arquivos grandes. Do ponto de vista da segurança, o Flash possuía vulnerabilidades no código, as quais permitiam que invasores infiltrassem código malicioso no computador dos usuários.

O ActiveX apresentado pela Microsoft não é executado em uma *sandbox*. Devido a isso, quando um controle ActiveX é instalado no computador do usuário, este torna-se parte do sistema operacional, e é capaz de adulterar a máquina [Golsch 2019]. Já o grande problema dos Java applets é que estes são executados automaticamente no navegador, sem o consentimento do usuário, tornando os recursos de segurança do Java insuficientes.

O JS é a linguagem mais utilizada por aplicações Web ao considerar a execução de conteúdo no lado do cliente [Yan et al. 2021]. Porém, o desempenho do JS é frequentemente considerado uma grande limitação na prática. Os programas JS são distribuídos em código fonte que precisa ser analisados e interpretados em tempo de execução. Os programas WebAssembly, por sua vez, são entregues como binários compilados, os quais podem ser carregados e executados mais rapidamente. Além disso, os programas WebAssembly geralmente são criados usando compiladores, os quais são responsáveis por compilar programas existentes em linguagens de programação de alto nível como C/C++ para o bytecode WebAssembly [Yan et al. 2021].

Das tecnologias predecessoras do WebAssembly é possível observar o uso de linguagens de alto nível como C/C++ na Web apenas no asm.js. Entretanto, com a criação do WebAssembly, o asm.js se tornou obsoleto, pois o WebAssembly possui um formato de *bytecode* que é mais rápido de analisar.

6.4. WebAssembly

O WebAssembly, também conhecido como Wasm, é um formato projetado para ser um alvo binário para linguagens de programação. A portabilidade permite aos desenvolvedores escolher a linguagem mais adequada para implementação de uma aplicação. Portanto, uma variedade de linguagens de alto nível não suportadas anteriormente no ambiente Web podem agora ser usadas, facilitando a tarefa dos desenvolvedores de software. Com linguagens de alto nível, os desenvolvedores têm mais flexibilidade, acesso a estruturas e um alvo binário compacto.

O *bytecode* WebAssembly é executado dentro de uma máquina virtual baseada em pilha. As aplicações podem ser executadas em um suporte de tempo de execução nativo (uma ferramenta de linha de comando) no lado do servidor, dentro do navegador Web ou até mesmo em dispositivos IoT. Da perspectiva do lado do cliente (como ilustrado na Figura 6.1), os binários compactos são recuperados de um servidor para serem executados em um ambiente *sandbox*, oferecendo um ganho de desempenho em relação a linguagens interpretadas como JS.

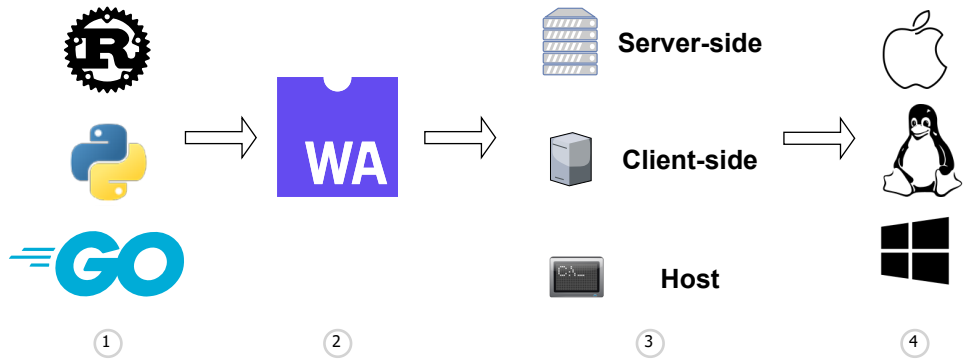

Figura 6.1. Processo de desenvolvimento em WebAssembly. (1) Um conjunto variado de linguagens pode ser utilizado para o desenvolvimento de aplicações (2) Compiladores específicos são utilizados para gerar os binários; (3) Estes binários podem ser executados em um conjunto variado de ambientes, inclusive (4) diferentes plataformas.

A Figura 6.2 apresenta a interação esperada para uma aplicação WebAssembly. Um binário WebAssembly é executado dentro de um ambiente sandbox, onde as instruções são adicionadas e retiradas da pilha, as informações são armazenadas na memória linear e recursos externos podem interagir com o ambiente *sandbox* por meio de uma função exposta [Rourke 2018]. Um exemplo de interação externa entre código Wasm e JS por meio de funções expostas é apresentado na Figura 6.2.

As instruções em uma máquina baseada em pilha assumirão que a maioria dos operandos está na pilha, em vez de nos registradores [Hoffman 2019]. A pilha WebAssembly é uma estrutura Last-In, First-Out (LIFO), qualquer interação e comandos com a pilha dentro do ambiente virtual serão limitados ao elemento no topo da pilha [Battagline 2021]. Essas opções de *design* permitem fácil portabilidade de outras linguagens e binários com tamanho pequeno [Hoffman 2019].

O formato WebAssembly suporta quatro tipos de dados: `i32` (número inteiro de

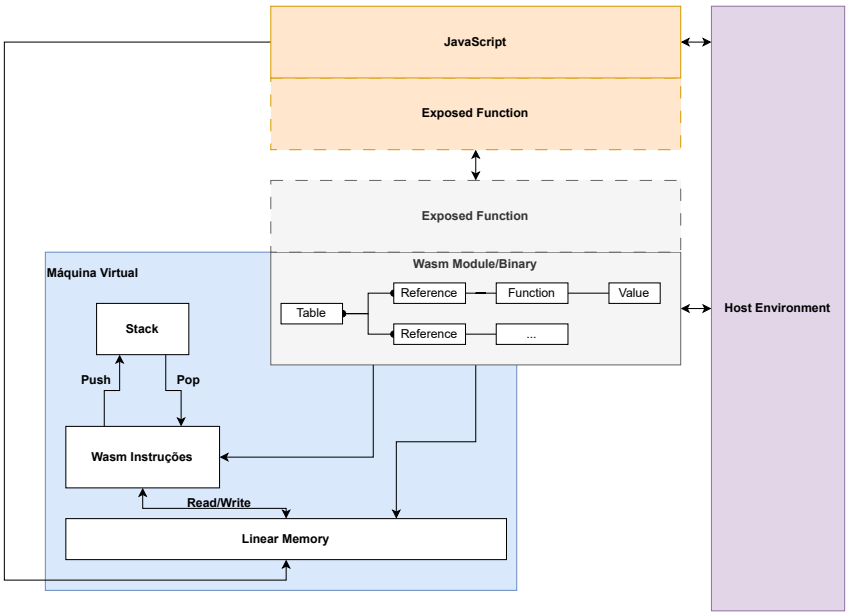

Figura 6.2. Exemplo de interação entre um módulo WebAssembly e recursos expostos.

32 bits), `i64` (número inteiro de 64 bits), `f32` (números de ponto flutuante de 32 bits) e `f64` (números de ponto flutuante de 64 bits). A sinalização intrínseca à representação numérica só é realizada quando uma operação é realizada, com operações específicas para dados com e sem sinal [Hoffman 2019]. *Strings* só podem ser usadas por meio de memória linear. O controle de *strings* e outros elementos encontrados na memória linear para recursos externos como JS é feito passando a posição e o comprimento desses elementos na memória [Battagline 2021]. Como o WebAssembly não possui *heap*, não há suporte a programação orientada a objetos. A memória linear é um bloco de bytes gerenciado pelo código Wasm; caso seja necessário mais espaço, os blocos podem ser incrementados em páginas, até um limite predefinido. O acesso à memória do *host* através da memória linear não é possível (exceto nos casos que seja definido explicitamente o acesso externo ao recurso).

A Figura 6.3 apresenta a estrutura de um binário WebAssembly. Ele é composto por um módulo WebAssembly, e estruturado em três seções: *Preâmbulo*, *Standard* e *Custom*. O *Preâmbulo* indica que o arquivo é um módulo WebAssembly e o formato da versão que está sendo usada. A seção *Standard* apresenta seções conhecidas que possuem funcionalidades específicas. A Tabela 6.3 mostra onze seções padrão. Nenhuma seção é exigida por um módulo Wasm, aparecendo apenas quando necessário. A seção *Custom* pode estar em qualquer posição de um módulo, permitindo a inclusão de dados que podem ser usados para depuração ou funções [Hoffman 2019].

6.4.1. WASI

A Application Programming Interface (API) padrão definida para desenvolvimento WebAssembly é conhecida como WebAssembly System Interface (WASI). Ela define as regras para o suporte de execução e as interações que o WebAssembly realiza com o ambiente. Através do WASI, o WebAssembly pode realizar operações nativas encontradas

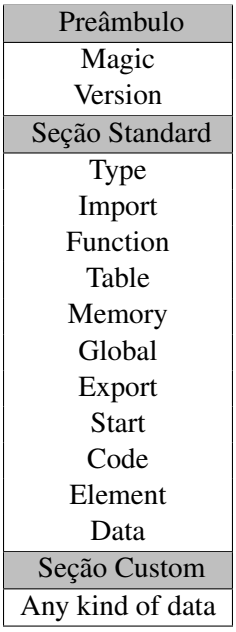

Figura 6.3. Estrutura de um binário WebAssembly (módulo WebAssembly).

em outras linguagens de programação, além de garantir os níveis de segurança definidos para o WebAssembly [Battagline 2021].

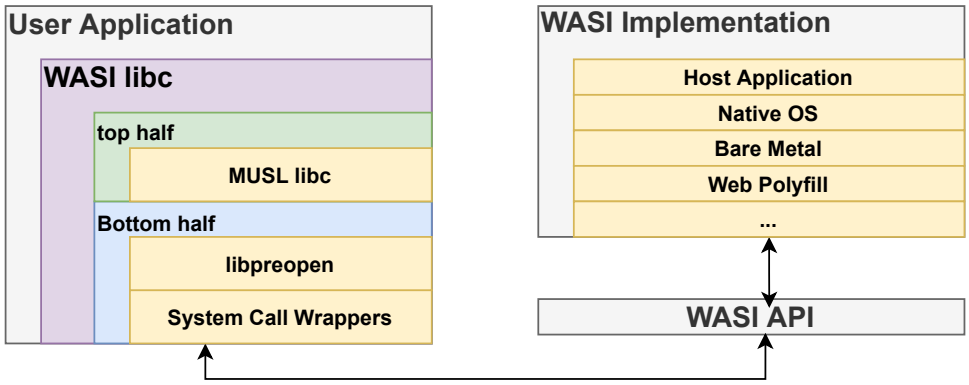

Figura 6.4. Arquitetura da WebAssembly System Interface (WASI) [BytecodeAlliance 2021a].

O foco inicial da WASI API é fornecer acesso a recursos como arquivos e rede. A WASI API usa WASI libc (biblioteca C padrão), que pretende ser uma interface libc. Esta implementação pode ser dividida em *bottom half*, que é composta por `libpreopen` e *wrappers* de chamadas de sistema; e *top half*, que é uma implementação da MUSL libc[1] [BytecodeAlliance 2021a]. A Figura 6.4 mostra esta arquitetura, com o fluxo de comunicação esperado. O *wrapper* de chamadas de sistema é responsável por fazer chamadas para a API WASI. Consequentemente, a API WASI faz as chamadas correspondentes à implementação WASI presente no sistema. Essas chamadas podem ser processadas pelo sistema operacional ou pelo suporte de tempo de execução de alguma linguagem.

[1]MUSL libc implementa a biblioteca padrão C sobre as chamadas do sistema Linux [Felker 2023].

Toda interação entre a aplicação e o sistema operacional é realizada através de chamadas WASI. Essas chamadas possuem funcionalidade semelhante às chamadas de sistema, pois sem elas um aplicativo não seria capaz de acessar os recursos do sistema.

6.4.2. Desenvolvimento e uso

O desenvolvimento de aplicações no formato WebAssembly pode ser realizado de duas formas: (i) o porte de aplicações desenvolvidas em outras linguagens como Rust, C/C++ e Go; ou (ii) através de uma representação legível chamada WebAssembly Text (WAT), que permite aos desenvolvedores escrever código WASM usando Expressões S ou estilos de lista de instruções lineares [Battagline 2021, W3C Community Group 2022b].

Emscripten [Emscripten 2021] e Wasmtime [BytecodeAlliance 2021b] são os dois compiladores mais populares para WebAssembly. Ambos seguem o padrão WASI mas têm objetivos diferentes. Emscripten é baseado na infraestrutura do compilador LLVM [LLVM 2023] e está focado na compilação de código C/C++ em WebAssembly. O principal alvo de execução são os navegadores da web, e os arquivos JS e HTML necessários são empacotados junto com o código *.wasm* gerado.

Wasmtime gera aplicativos *.wasm* que podem ser executados no navegador ou como um aplicativo no *host* sem a necessidade de um mecanismo de navegador. Devido a essas características, o Wasmtime fornece uma gama de ferramentas de depuração para avaliar a interação do código com o host e também fornece uma maneira de recuperar chamadas WASI do tempo de execução de uma aplicação.

Um total de 45 chamadas já estão implementadas no Wasmtime; estas são as chamadas que seguem o padrão WASI (vamos nos referir a essas chamadas por *chamadas WASI*). Wasmtime segue os *namespaces* (*snapshots*) definidos para a WASI API para controlar o suporte de recursos do compilador e, consequentemente, também controla o tipo de chamadas WASI suportadas em uma versão específica. A versão atual 19.0.2 do Wasmtime define as chamadas WASI no *snapshot_preview1*, e futuras versões de *snapshot* permitirão que os desenvolvedores usem funções de um *snapshot* diferente, o que consequentemente fornecerá mais chamadas.

Em geral, os compiladores para WebAssembly estão em estágios iniciais de desenvolvimento e as mudanças são frequentes. A vantagem do compilador Wasmtime está na forma como as chamadas WASI são tratadas, com a possibilidade de executar e observar essas aplicações sem a necessidade de utilizar um navegador.

WebAssembly possui uma comunidade ativa que está constantemente implementando melhorias e novos recursos. A versão 1.0 do WebAssembly não suportava um coletor de lixo para gerenciamento de memória até o final de 2023 e não fornece acesso direto ao Document Object Model (DOM) [Rourke 2018].

Uma aplicação WebAssembly (ou um arquivo WAT) contém um módulo, que será composto por funções e variáveis (seguindo a estrutura apresentada na Tabela 6.3). As operações de exportação e importação fornecem acesso às funções do WebAssembly. A operação de exportação disponibiliza uma função Wasm para outras operações no ambiente (por exemplo, um valor compartilhado JS durante o processo de execução). As funções no ambiente também podem ser importadas para o WebAssembly usando a ope-

ração de importação.

Visando a melhor compreensão do formato, as Listagens 6.1 e 6.2 apresentam uma implementação de uma função *Hello, world!* em Rust e WebAssembly (usando o formato WAT) respectivamente. A implementação em Rust consiste de uma função *main* que usa a operação *println* para imprimir a mensagem na tela do usuário (linhas 1–3). O código é compilado usando *rustc* (linha 5), e o binário gerado é executado na linha de comando (linhas 6 e 7).

```
1 fn main() {
2    println!("Hello, world!");
3 }
4
5 $ rustc main.rs
6 $ ./main
7 Hello, world!
```

Listagem 6.1. Exemplo de uma implementação de *Hello, world!* em Rust.

A versão WebAssembly é diferente devido à limitação de tipos do formato. Tipos como *strings*, objetos ou dicionários, que podem ser encontrados em muitas linguagens de programação de alto nível, não são suportadas pelo formato WebAssembly [Sletten 2022]. Desta forma, vamos utilizar este exemplo para explanar o formato e as principais operações efetuadas durante o processo de execução.

O formato WAT adota uma representação similar à de linguagens como Lisp. Linhas iniciadas por `;;` são comentários. O início de uma aplicação é definido pela palavra reservada `module` que define o início de uma aplicação, como toda a estrutura que será utilizada pelo sistema para carregar os recursos necessários para executar a aplicação, seguindo os campos estruturais apresentados na Figura 6.3. O código do exemplo (Listagem 6.2) está organizado internamente em sete partes:

(1) A aplicação WebAssembly pode precisar de acesso a recursos externos para realizar suas operações corretamente. A palavra chave `import` realiza a importação desses recursos. Neste caso, o módulo requer a função `fd_write()`, encontrada no *namespace* `wasi_unstable`. Essa função permite escrever um ou mais vetores de bytes em um arquivo, e é usada no exemplo para enviar uma *string* para a saída padrão (*stdout*). `fd_write()` recebe quatro parâmetros: (i) o descritor de arquivo; (ii) início do primeiro vetor na memória linear (deslocamento em bytes); (iii) número de elementos do vetor; (iv) o deslocamento em bytes na memória linear onde a função retorna o número de bytes escritos.

(2) Como esta informação precisa ser escrita em algum lugar, declaramos a memória linear (linha 6). Uma função dentro do módulo só se torna visível no momento que ela é exportada (`export`), desta forma exportamos a memória, já que a função `fd_write()` que foi importada vai precisar interagir com a memória.

(3) Agora que temos acesso à memória e à função de escrita, utilizamos o elemento `data` para escrever a *string Hello, world!* na memória. Como o segundo argumento de `fd_write()` são vetores de bytes, a memória fica organizada da seguinte forma:

- bytes 0–3: deslocamento da *string* (em relação ao início do vetor, isto é, ao byte 0);
- bytes 4–7: comprimento da *string* (em bytes);
- bytes 8–19: a *string* propriamente dita; e
- bytes 20–23: espaço que será usado para armazenar o número de bytes escritos por `fd_write()`.

Logo, a *string* é armazenada em um deslocamento de 8 bytes em relação ao início da memória linear.

(4) Para juntar todas estas operações/funções, exportamos uma nova função `main`.

(5) A primeira operação realizada pela função `main` consiste em criar o vetor correspondente à *string*. Declaramos um ponteiro para o início da *string* (linha 15), considerando o nosso deslocamento de 8 bytes, e que a *string* possui tamanho de 12 bytes (linha 16).

(6) Realizamos a chamada de `fd_write()`. O descritor de arquivo é 1, que corresponde à saída padrão (linha 20). O vetor a ser escrito está na posição 0 da memória linear (linha 21). O número de elementos desse vetor é 1 (linha 22). O número de bytes escritos por `fd_write()` será armazenado na posição 20 da memória linear (linha 23).

(7) Descartamos os *bytes* escritos no topo da pilha.

Resultado: Por fim, podemos observar a execução do módulo WebAssembly utilizando o *wasmtime*. Neste caso por utilizarmos o *wasmtime* não é necessário compilar e executar, já que o compilador realiza todas estas operações automaticamente.

```
1  (module
2     ;; (1)
3     (import "wasi_unstable" "fd_write" (func $fd_write (param i32 i32 i32 i32) (
          result i32)))
4
5     ;; (2)
6     (memory 1)
7     (export "memory" (memory 0))
8
9     ;; (3)
10    (data (i32.const 8) "Hello, world!\n")
11
12    ;; (4)
13    (func $main (export "_start")
14       ;; (5)
15       (i32.store (i32.const 0) (i32.const 8))
16       (i32.store (i32.const 4) (i32.const 12))
17
18       ;; (6)
19       (call $fd_write
20          (i32.const 1)
21          (i32.const 0)
22          (i32.const 1)
23          (i32.const 20)
24       )
25       ;; (7)
```

```
      drop
   )
)

$ curl https://wasmtime.dev/install.sh -sSf | bash
$ wasmtime main.wat
Hello, world!
```

Listagem 6.2. Exemplo de *Hello, world!* em WebAssembly [BytecodeAlliance 2024].

O formato WebAssembly pode ser utilizado por outras linguagens para a execução de funcionalidades específicas. Por exemplo, uma aplicação implementada em Rust pode invocar uma função em WebAssembly. Para explanar este caso, apresentamos a Listagem 6.3, que é um código Rust que imprime o máximo divisor comum (mdc) de dois números (6 e 27); o cálculo em si é efetuado por uma função `gcd()`, implementada em WebAssembly. O código Rust envolve os seguintes passos:

(1) Importamos a *crate* necessária para realizar estas operações.

(2) Para que a aplicação em Rust consiga acessar os recursos da função exposta no módulo WebAssembly, precisamos conseguir invocar o módulo (linha 8). Para alcançar este ponto, precisamos realizar a análise sintática e armazenar este módulo (linha 7).

(3) Uma referência para a função `gcd` do módulo WebAssembly exposto é armazenada em `gcd`.

(4) Invocamos a função usando `gcd.call()` e imprimimos o resultado.

```rust
// (1)
use wasmtime::*;

fn main() -> Result<()> {
   // (2)
   let mut store = Store::<()>::default();
   let module = Module::from_file(store.engine(), "examples/gcd.wat")?;
   let instance = Instance::new(&mut store, &module, &[])?;

   // (3)
   let gcd = instance.get_typed_func::<(i32, i32), i32>(&mut store, "gcd")?;

   // (4)
   println!("gcd(6, 27) = {}", gcd.call(&mut store, (6, 27))?);
   Ok(())
}
```

Listagem 6.3. Código Rust que usa uma função Wasm para calcular o máximo divisor comum (mdc) [BytecodeAlliance 2024].

A função WebAssembly (Listagem 6.4) calcula o mdc usando o algoritmo de Euclides. O código (linha 2) define a função `gcd`, que recebe dois parâmetros inteiros e retorna o mdc (também um inteiro). Esta função é exposta (linha 26) para permitir as interações com o ambiente externo.

```
(module
  (func $gcd (param i32 i32) (result i32)
```

```
3     (local i32)
4     block ;; label = @1
5       block ;; label = @2
6         local.get 0
7         br_if 0 (;@2;)
8         local.get 1
9         local.set 2
10        br 1 (;@1;)
11      end
12      loop ;; label = @2
13        local.get 1
14        local.get 0
15        local.tee 2
16        i32.rem_u
17        local.set 0
18        local.get 2
19        local.set 1
20        local.get 0
21        br_if 0 (;@2;)
22      end
23    end
24    local.get 2
25  )
26  (export "gcd" (func $gcd))
27 )
```

Listagem 6.4. Função GCD em WebAssembly [BytecodeAlliance 2024].

Agora vamos considerar como o formato WebAssembly seria utilizado junto com o JS. Para realizar esta tarefa, precisamos de uma implementação de *Hello, world!* em WebAssembly (Listagem 6.5) que será compilada para gerar um arquivo *.wasm*; um aplicação em JS que será responsável por preparar o ambiente e invocar a aplicação em WebAssembly (Listagem 6.6); e um arquivo *.html* para definir um ambiente mínimo (Listagem 6.7) [Apodaca 2020].

O código do arquivo *hello.wat* pode ser observado na Listagem 6.5. As linhas 3 e 4 importam as dependências criadas no lado do JS. Em seguida a linha 7 escreve o texto *"Hello, World!"*, começando no índice de memória 0 usando o operador de dados. Por fim, é exportada uma função chamada `hello` na linha 10. Dentro desta função, dois valores são inseridos sequencialmente (0 e 13). O primeiro é um deslocamento para a memória compartilhada definida no lado do JS. O segundo é o tamanho da *string "Hello, World"*. O operador `call` retira os dois valores da pilha e invoca a função `log()` com esses dois parâmetros.

```
1  (module
2   ;; (1)
3   (import "env" "memory" (memory 1))
4   (import "env" "log" (func $log (param i32 i32)))
5
6   ;; (2)
7   (data (i32.const 0) "Hello, World!")
8
9   ;; (3)
10  (func (export "hello")
11    ;; (4)
12    i32.const 0
13    i32.const 13
14    call $log
15  )
16 )
```

Listagem 6.5. Exemplo de *Hello, world!* [Apodaca 2020].

A Listagem 6.6 apresenta como esta interação vai ocorrer no lado do JS. Este processo segue a lógica apresentada na Listagem 6.3, onde a aplicação em JS precisa invocar o binário WebAssembly, neste caso precisa ser invocada a função `log()`. Esta função vai acessar a memória compartilhada, que foi exposta no módulo WebAssembly, para ler a *string* de acordo com os parâmetros de posição e tamanho.

```
const memory = new WebAssembly.Memory({ initial: 1 });

const log = (offset, length) => {
  const bytes = new Uint8Array(memory.buffer, offset, length);
  const string = new TextDecoder('utf8').decode(bytes);

  console.log(string);
};

(async () => {
  const response = await fetch('./hello.wasm');
  const bytes = await response.arrayBuffer();
  const { instance } = await WebAssembly.instantiate(bytes, {
    env: { log, memory }
  });

  instance.exports.hello();
})();
```

Listagem 6.6. Função em JS para invocar o binário WebAssembly [Apodaca 2020].

Na Listagem 6.7 apresentamos o HTML que será encontrado no servidor Web, e que chama o *script* em JS para ser executado.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Hello, World! in WebAssembly</title>
  </head>
  <body>
    <script src="hello.js" type="module"></script>
  </body>
</html>
```

Listagem 6.7. HTML da página [Apodaca 2020].

O último exemplo das funcionalidades do WebAssembly consiste em portar aplicações de uma linguagem específica para Wasm. A Listagem 6.8 apresenta uma função em Rust que será portada para WebAssembly. Esta função recebe dois valores `usize` e realiza a soma dos dois valores, retornando um `usize`. Usamos `no_mangle` para que os nomes de funções sejam mantidos para posterior análise, e `extern` para que o compilador trate a função como estejam utilizando a *C ABI*. O alvo de compilação `wasm32-unknown-unknown` é usado para uma função que será executada dentro de um ambiente JS, enquanto o alvo `wasm32-wasi` é usado para código executado no seu próprio *runtime*. Desta forma, o resultado deste processo de compilação consiste em um binário que pode ser executado, por exemplo, em um ambiente Web similar ao da Listagem 6.5. Caso seja desejado executar o código em Rust apresentado na Listagem 6.1 com o *wasmtime*, só é necessário realizar a troca do alvo para `wasm32-wasi` como apresentado na Listagem 6.9.

```
#[no_mangle]
pub extern "C" fn add(value_a: usize, value_b: usize) -> usize {
    value_a + value_b
}

$ rustup target add wasm32-unknown-unknown
$ cargo build --target=wasm32-unknown-unknown
$ ls target/wasm32-unknown-unknown/debug/add_wasm.wasm
```

Listagem 6.8. Exemplo de função em Rust que será compilada para WebAssembly [Levick 2024].

```
$ rustup target add wasm32-wasi
$ rustc hello.rs --target=wasm32-wasi
$ wasmtime hello.wasm
Hello, world!
```

Listagem 6.9. Execução da função Rust usando *wasmtime* [Levick 2024].

Agora podemos entender melhor o conteúdo encontrado no binário (o nosso módulo WebAssembly que foi gerado a partir de um código em Rust). Vamos utilizar o *wasm-tools*[2] para inspecionar os binários. Ao executar um *wasm-tools print –skeleton target/wasm32unknownunknown/debug/add_wasm.wasm* conseguimos entender a estrutura encontrada no módulo WebAssembly, como apresentado na Listagem 6.10. Ao comparar estruturas anteriores, encontramos o `export` da memória linear (linha 29), bem como o da função `add` (linha 30) criada na Listagem 6.8 e a definição da nossa função `add` (linha 15), que recebe dois parâmetros.

```
(module
  (type (;0;) (func (param i32 i32)))
  (type (;1;) (func (param i32 i32 i32) (result i32)))
  (type (;2;) (func (param i32 i32) (result i32)))
  (type (;3;) (func (param i32 i32 i32)))
  (type (;4;) (func (param i32 i32 i32 i32) (result i32)))
  (type (;5;) (func (param i32)))
  (type (;6;) (func (param i32 i32 i32 i32)))
  (type (;7;) (func (param i32) (result i32)))
  (type (;8;) (func (param i32 i32 i32 i32 i32 i32)))
  (type (;9;) (func))
  (type (;10;) (func (param i32 i32 i32 i32 i32 i32) (result i32)))
  (type (;11;) (func (param i32 i32 i32 i32 i32) (result i32)))
  (type (;12;) (func (param i64 i32 i32) (result i32)))
  (func $add (;0;) (type 2) (param i32 i32) (result i32) ...)
  (func $__rust_alloc (;1;) (type 2) (param i32 i32) (result i32) ...)
  (func $__rust_dealloc (;2;) (type 3) (param i32 i32 i32) ...)
  (func $__rust_realloc (;3;) (type 4) (param i32 i32 i32 i32) (result i32) ...)
  (func $__rust_alloc_error_handler (;4;) (type 0) (param i32 i32) ...)
  (func $_ZN36_$LT$T$u20$as$u20$core..any..Any$GT$7type_id17h2a84f20e718440dfE
      (;5;) (type 0) (param i32 i32) ...)
...
  (func $_ZN17compiler_builtins3mem6memcpy17h7037a3a0dead1e85E (;54;) (type 1) (
      param i32 i32 i32) (result i32) ...)
  (func $memcpy (;55;) (type 1) (param i32 i32 i32) (result i32) ...)
  (table (;0;) 20 20 funcref)
  (memory (;0;) 17)
  (global $__stack_pointer (;0;) (mut i32) i32.const 1048576)
  (global (;1;) i32 i32.const 1049701)
  (global (;2;) i32 i32.const 1049712)
  (export "memory" (memory 0))
  (export "add" (func $add))
```

[2]Wasm-tools pode ser instalado com *cargo install wasm-tools*, instruções em `https://github.com/bytecodealliance/wasm-tools`.

```
31  (export "__data_end" (global 1))
32  (export "__heap_base" (global 2))
33  (elem (;0;) (i32.const 1) ...)
34  (data $.rodata (;0;) (i32.const 1048576) ...)
35  (@custom ".debug_abbrev"
36  ...
```

Listagem 6.10. Exemplo da estrutura encontrada no WebAssembly compilado do Rust [Levick 2024] (parte do *output* foi omitida).

Na prática, o WebAssembly é usado para criptografia, vídeo, áudio, gráficos e outras atividades que possam aproveitar as vantagens do *design* do formato. No entanto, o formato WebAssembly não se limita ao navegador da web. Aplicativos do lado do servidor, serviços distribuídos e aplicativos móveis também podem aproveitar as vantagens do WebAssembly [Rossberg 2024].

A pesquisa sobre o formato WebAssembly no primeiro ano da proposta de desenvolvimento é bastante limitada, pois o foco estava na definição do formato. A primeira visão geral do formato é apresentada por [Haas et al. 2017]. Esta é a primeira discussão aprofundada sobre funcionalidades e opções de *design*. O artigo faz uma excelente abordagem para apresentar esta nova tecnologia, discutindo os conceitos por trás do processo de validação, execução e respectivas restrições. Sendo um dos primeiros textos sobre WebAssembly, o trabalho ajuda a comunidade com uma rica documentação e orienta numerosos trabalhos.

Algumas das principais *features* encontradas no WebAssembly em 2024 são:

- *Mutable globals*: agora variáveis *const* podem ser importadas ou exportadas, oferecendo um ambiente mais flexível para o desenvolvimento de aplicações, como, por exemplo, aplicações *multithreaded*.
- *Multiple memories*: módulos podem declarar várias memórias; no entanto, módulos e funções só podem fazer referência a uma memória por vez, impossibilitando a transferência de dados entre as memórias.
- Tratamento de exceções, tanto aquelas lançadas em módulos Wasm quanto por funções em outras linguagens que são importadas e invocadas em código WebAssembly.
- Coletor de lixo para gerenciamento de memória, que já foi adotada no Mozilla Firefox e no Google Chrome.

6.5. Aspectos de desempenho

O formato WebAssembly já é suportado pelos principais navegadores, trazendo um novo recurso para o ambiente Web ao permitir que aplicações desenvolvidas em C, C++, C#, Go e Rust sejam portadas para este ambiente [Jangda et al. 2019]. Um dos atrativos do WebAssembly é o potencial de gerar ganhos de desempenho em comparação com o JS. Diante disso, esta seção discute alguns aspectos de desempenho do WebAssembly.

Em um navegador, tanto aplicações WebAssembly quanto aplicações JS são executadas no *engine* JS. As principais diferenças seriam o modelo de execução e o controle

da memória. Uma aplicação em JS precisa passar por uma análise sintática, otimizada e compilada antes de poder ser executada. Um compilador Just-In-Time (JIT) é responsável por estas operações de otimização e geração de *bytecode* que permitem o aperfeiçoamento de códigos que serão executados várias vezes. Ao considerar a memória no JS, todo o controle é realizado pelo coletor de lixo (Garbage Collection (GC)), que realiza a identificação de blocos que não serão mais utilizados e podem ser desalocados.

Já para o formato WebAssembly o *bytecode* já está pronto, sem necessidade de realizar análise sintática ou otimização do código, já que essas etapas foram feitas durante a geração do *bytecode*. O WebAssembly armazena as informações em uma memória linear onde as informações podem ser compartilhadas entre a aplicação WebAssembly e aplicações no ambiente externo, como, por exemplo, JS. Através da memória linear, os dados são passados para um módulo WebAssembly, e as informações são compartilhadas entre JS e WebAssembly, agindo como um vetor onde os dados são armazenados. A memória é alocada através de páginas, e após alocada não pode ser desalocada. Todo o controle das informações contidas na memória linear é responsabilidade do desenvolvedor, que deve rastrear o que está sendo armazenado na memória e onde está [Battagline 2021]; quando uma linguagem de alto nível é compilada para Wasm, o compilador insere código para efetuar esse controle. Recentemente foi lançado o suporte de GC para o WebAssembly, no entanto, discussões do impacto ainda são limitadas. Versões anteriores sem o suporte do GC requerem que o desenvolvedor faça todo o processo de controle da memória linear, devido à inexistência de uma estratégia automática para recuperar memória [Yan et al. 2021].

Ao considerar o uso de memória, aplicações desenvolvidas em JS tiram o proveito do GC, que tem um direto impacto na memória utilizada pela aplicação; isso difere de aplicações WebAssembly, que possuem um uso de memória superior [Wang 2021]. Estas diferenças tornam-se consideráveis quando as entradas de dados começam a ser maiores: para o JS o uso de memória permanece constante, enquanto que as aplicações Wasm exibem crescimento do uso de memória. [Wang 2021] demonstra este comportamento com cinco tamanhos de entradas variando de pequeno a extra-grande, onde o JS mantém uma média de uso de memória de 884,42 KB contra um uso médio de 31,2 MB do WebAssembly.

Ao avaliar o tempo de execução entre WebAssembly e JS, é possível observar que o JS é impactado pelo uso de JIT, ao passo que o WebAssembly mantém um desempenho mais constante [Wang 2021, Yan et al. 2021]. Em um estudo [De Macedo et al. 2022], o WebAssembly foi 9,8% mais lento que o JS em avaliações com *benchmarks*, e foi 17,2% mais rápido na execução de aplicações reais.

O consumo de energia é um dos fatores de relevância em novas tecnologias, e o comportamento observado acaba sendo similar em relação às comparações de desempenho. Aplicações reais em WebAssembly consumiram 24,3% menos energia que aplicações em JS; no entanto, ao considerar testes com *benchmarks*, o WebAssembly gastou 5,1% mais energia que o JS [De Macedo et al. 2022].

Apesar do WebAssembly ser uma tecnologia recente, ele já está disponível em todos os principais navegadores, sendo importante as diferenças entre eles. O WebAssembly apresenta os melhores resultados ao considerar consumo energético e tempo de

execução com o Google Chrome, ao passo que com o Mozilla Firefox e Microsoft Edge os resultados variam de acordo com o tipo de aplicação [De Macedo et al. 2022]. Em uma comparação entre ambientes *desktop* e móveis [Yan et al. 2021], o Mozilla Firefox foi o navegador mais rápido para executar aplicações em JS em dispositivos móveis, e o Microsoft Edge foi o mais eficiente para WebAssembly.

Um dos *overheads* encontrados ao utilizar o WebAssembly ocorre em momentos que a aplicação requer uma troca de contexto. Ou seja, quando a aplicação está trocando o contexto entre o WebAssembly e o JS, para acessar um tipo de recurso como o DOM ou *WebSockets*. Isto ocorre em momentos que o WebAssembly necessita acessar APIs Web, já que estas operações requerem no mínimo instanciar o módulo WebAssembly [Yan et al. 2021].

[De Macedo et al. 2021] destaca que o WebAssembly foi mais rápido e teve menor consumo de energia na maioria dos testes realizados em comparação com o JS. No entanto, o trabalho também destaca que, quanto maior a entrada de dados da aplicação, menores são as diferenças de desempenho entre o WebAssembly e o JS. Em suma, apesar do WebAssembly ainda requerer melhorias para superar algumas limitações, em muitos cenários essa tecnologia já propicia ganhos de desempenho e de consumo de energia em comparação com o JS.

6.6. Aspectos de segurança em WebAssembly

No geral, a pesquisa do WebAssembly está sendo desenvolvida com diversos propósitos. As discussões atuais concentram-se na exploração da segurança desta tecnologia, na identificação de falhas de *design*, no uso deste formato com intenções maliciosas e na aplicação dos benefícios desta tecnologia a outros recursos. Além disso, atacantes também se utilizam do WebAssembly para realização de ataques, principalmente *cryptojacking* e ofuscação de códigos [Helpa et al. 2023, Heinrich et al. 2023, Heinrich et al. 2024].

Portanto, nesta seção vamos discutir como o *sandbox* do WebAssembly funciona (Seção 6.6.1), como a integridade do fluxo de controle traz mais segurança para o formato (Seção 6.6.2), como atacantes tentam explorar essa nova tecnologia e quais vulnerabilidades/riscos ainda existem e estão sendo estudados pela comunidade para a criação de contramedidas (Seção 6.6.3).

6.6.1. Sandbox WebAssembly

Uma *sandbox* é um ambiente de execução restrito e controlado que garante que *software* potencialmente malicioso, como código móvel, acesse somente os recursos do sistema para os quais esteja explicitamente autorizado [Shirey 2007]. Seja no navegador ou em um suporte de execução nativo, código WebAssembly é executado isolado dentro de uma *sandbox*, que concede acesso externo apenas através de uma API. Este *design* permite a rápida execução de aplicações no ambiente web, uma vez que a aplicação está pronta para ser executada após o *download* do *bytecode*. Para isso, o compilador (ou interpretador) insere verificações de tempo de execução que restringem o código à sua própria região de memória. Uma aplicação só poderá acessar informações fora da *sandbox* se a permissão correta for concedida. O módulo WebAssembly segue o Same Origin Policy (SOP) e o acesso a recursos específicos também requer o uso de uma API

[Kim et al. 2022, Johnson et al. 2023].

O Software-fault Isolation (SFI) *sandbox* permite que cada instância seja executada em seu próprio ambiente *sandbox* isolado. Desta forma, desde que o ambiente não tenha vulnerabilidades, aplicações podem ser executadas sem ameaçar a segurança do ambiente [Bosamiya et al. 2022]. Ao considerar a memória linear, ela não compartilha o mesmo espaço do código, estrutura e pilha, não podendo ser usada para corromper o ambiente de execução [Haas et al. 2017].

6.6.2. Integridade do Fluxo de Controle

Diferentemente de código nativo ou *bytecode* Java, WebAssembly possui apenas fluxo de controle estruturado (Structured Control Flow (SCF)), com as instruções em uma função organizadas em blocos bem aninhados. As aplicações devem declarar todas as funções acessíveis e seus tipos associados no momento da inicialização, mesmo quando são utilizadas bibliotecas de vinculação dinâmica. Assim, o WebAssembly garante a integridade do fluxo de controle por meio da semântica de execução do próprio formato, definindo construções de código válidas e como o fluxo de controle só pode pular para o início de uma construção válida [Dejaeghere et al. 2023, Lehmann et al. 2020].

A semântica de execução garante implicitamente a segurança de chamadas diretas por meio do uso de índices de seção de função explícitos. Instruções do tipo `goto` ou saltos para endereços arbitrários não são possíveis. Já a segurança do retorno da função é garantido por meio de uma pilha de chamadas protegida. A assinatura de tipo de chamadas de função indiretas já é verificada em tempo de execução, implementando efetivamente a integridade do fluxo de controle baseada em tipo de granularidade grossa para chamadas de função indiretas [Lehmann et al. 2020].

Ataques diretos de injeção de código não são possíveis em WebAssembly, já que não se pode executar dados na memória como instruções de *bytecode*. Entretanto, um atacante pode sequestrar o fluxo de controle de um módulo WebAssembly usando ataques de reutilização de código contra chamadas indiretas. Ressalta-se que ataques do tipo Return-Oriented Programming (ROP) convencionais usando sequências curtas de instruções não são possíveis em WebAssembly, porque a integridade do fluxo de controle garante que os alvos de chamada sejam funções válidas declaradas no tempo de carga. Entretanto, condições de corrida, como vulnerabilidades do tipo *time of check to time of use* (TOCTTOU), são possíveis no WebAssembly, uma vez que nenhuma garantia de execução ou agendamento é fornecida. Também podem ocorrer ataques *side-channel*, como ataques de temporização contra módulos.

6.6.3. Vetores de ataques e mitigações

Do ponto de vista de um invasor, o WebAssembly pode ser explorado para realizar uma ampla gama de estratégias de ataque. Nesta seção, focamos em estudos que consideram o uso do WebAssembly como vetor de ataque.

O primeiro estudo em grande escala para avaliar a presença e popularidade do WebAssembly na Internet foi realizado por [Musch et al. 2019]. Ele apontou que 50% das páginas identificadas como usando WebAssembly acabaram explorando esse recurso

para alguma atividade maliciosa (no top 1 milhão do Alexa[3], 56% das páginas estavam usando WebAssembly para fins maliciosos, como ofuscação ou *cryptojacking*) e apenas 10% das páginas observadas tinham códigos únicos. Portanto, antes de mergulhar nas falhas e vulnerabilidades do WebAssembly, discutiremos os tipos de vetores de ataque explorados pelos invasores.

Com foco nos binários WebAssembly (*.wasm*) que podem ser encontrados na Internet, o artigo [Hilbig et al. 2021] coletou e avaliou as propriedades de segurança de 8,4 k binários. Eles identificaram que binários gerados por linguagens que não possuem garantias de segurança na memória podem exportar essas vulnerabilidades para WebAssembly; a importação de API potencialmente perigosa existia em 21% dos binários observados, e 65% dos binários observados estavam usando uma parte da memória linear que não era gerenciada.

6.6.3.1. *Cryptojacking*

Cryptojacking é um ataque que visa explorar os recursos computacionais das vítimas para minerar criptomoedas. O WebAssembly é atrativo para esse tipo de operação devido ao ganho de desempenho em comparação a outras linguagens Web, uma vez que o desempenho impacta diretamente nos ganhos obtidos com esse tipo de ataque.

WebEth [Tiwari et al. 2018] é uma arquitetura distribuída de *cryptojacking* que usa JS e WebAssembly para mineração de Ethereum em um conjunto de navegadores. Sem a exigência de dependências externas, a implementação lida com a restrição de memória e rede imposta aos navegadores. Apesar do desempenho ser 30% mais lento que uma implementação em C++, a técnica *lazy* adotada é a principal contribuição do trabalho.

[Bian et al. 2019, Bian et al. 2020] propõem o *MineThrottle*, uma ferramenta que usa funcionalidades do WebAssembly e de algoritmos conhecidos de *cryptojacking* para identificar e interromper a execução deste tipo de ataque. Uma contribuição do trabalho é a discussão de possíveis estratégias para identificação deste tipo de ataque, apresentando as estratégias existentes e seus pontos fortes e fracos. As estratégias para detecção de *cryptojacking* na realidade utilizam um perfil para penalizar o processo de mineração que poderia estar em execução. A avaliação usa Alexa top 100 mil e 1 milhão. A estratégia de impressão digital atingiu um falso negativo/positivo inferior a 2%.

Seismic é um sistema de detecção que avisa os usuários quando uma atividade de *cryptojacking* é detectada [Wang et al. 2018]. A detecção baseada em assinatura é facilmente evitada por um invasor com o uso de estratégias de ofuscação. O *Seismic* utiliza recursos de código semântico para o processo de detecção, atingindo uma precisão de 98% para quatro famílias de algoritmos de *cryptojacking* em WebAssembly.

MinerGate é uma defesa baseada em aprendizado de máquina contra *cryptojacking* [Yu et al. 2020], que usa uma extensão de *proxy* para proteger o navegador. A extensão foi treinada com um conjunto de algoritmos de mineração em WebAssembly e é capaz de identificar elementos de interação de uma atividade de *cryptojacking* na rede.

[3]O ranking Alexa consiste em um conjunto de dados onde os sites são classificados de acordo com seu tráfego [Le Pochat et al. 2019].

[Petrov et al. 2020] apresenta *CoinPolice*, um método de detecção baseado em um classificador de rede neural profunda. Os recursos utilizados para o modelo consistem em características de desempenho e padrões de execução. A proposta detectou 97,8% dos mineradores e teve um falso positivo de 0,7%. No conjunto de dados reais, foram identificadas 6,7 mil páginas com mineradores ocultos.

6.6.3.2. Ofuscação

Ofuscação visa a transformação do código fonte ou binário com o objetivo de alterar sua aparência. Isto pode ser usado para a proteção da propriedade intelectual do código ou para ocultar código com fins maliciosos [Balakrishnan and Schulze 2005]. Como o WebAssembly é uma nova tecnologia na Web, usuários maliciosos podem tentar usá-lo para contornar medidas de segurança por meio de ofuscação de código.

[Romano et al. 2022] apresenta uma nova estratégia para escapar dos processos clássicos de identificação já propostos para JS através do uso de ofuscação de código. *Wobfuscator* é uma ferramenta de ofuscação que visa utilizar WebAssembly para movimentar parte do processamento que seria realizado em JS. Desta forma, a aplicação ainda atingirá seu objetivo, mas com pontos-chave da implementação sendo migrados para WebAssembly é possível fugir das soluções de análise estática já propostas para identificação de ataques em JS.

Um estudo do potencial de ofuscação através do uso do WebAssembly é o foco de [Bhansali et al. 2022]. Através de uma série de amostras benignas e maliciosas, diferentes estratégias de ofuscação foram aplicadas e testadas contra um detector de *cryptojacking*. Os resultados mostraram uma oportunidade altamente eficaz no uso de ofuscação para evitar engenharia reversa e descompilação do binário Wasm. O sucesso da ofuscação é influenciado pelo tipo de aplicação e pela complexidade do código, onde o código com características específicas dificultam o processo de ofuscação.

[Wang et al. 2019] apresenta uma solução para ofuscar operações numéricas intensivas encontradas em JS. Este sistema, denominado *JSPro*, consiste em uma virtualização de código para JS construído em WebAssembly. No entanto, o processo de tradução é limitado e, em alguns casos, os aplicativos não puderam ser traduzidos corretamente para o WebAssembly.

[Arteaga et al. 2021] aborda a diversificação de código para binários WebAssembly. Esta estratégia visa a geração de diferentes binários para um mesmo programa. O estudo apresenta *CROW*, um *framework* para diversificação de código em WebAssembly. A solução permite a diversificação de binários e rastreamentos para um programa. O *framework* é responsável por automatizar fluxos de trabalho para LLVM[4]. Os resultados mostram que a proposta foi capaz de alcançar diversidade ao gerar código para 79% do total de amostras no conjunto de dados utilizado.

[4]O projeto LLVM oferece um conjunto de compiladores e conjuntos de ferramentas que podem ser usados para o desenvolvimento de qualquer linguagem de programação [LLVM-Team 2023].

6.6.3.3. *Write Primitive*

Write Primitive explora vulnerabilidades encontradas em linguagens, como C, que podem permitir que um invasor obtenha uma primitiva de gravação no WebAssembly. Em alguns casos, esta vulnerabilidade não está presente na linguagem de origem, mas aparece quando o código é portado (compilado) para Wasm [Lehmann et al. 2020]. Na literatura, algumas vulnerabilidades utilizadas para alcançar esse propósito são:

- *Integer overflows/underflows:* para armazenar variáveis, o formato WebAssembly suporta tipos numéricos estáticos, semelhantes a linguagens como C/C++ e Rust. Um tipo esperado precisa ser definido com a variável que armazenará o valor. O uso de tipagem estática oferece melhor desempenho, pois não há necessidade de rastrear os tipos de variáveis durante a execução. Linguagens como Python e JS possuem tipagem dinâmica, onde o tipo de uma variável é assumido durante a execução (desta forma o desenvolvedor não precisa se preocupar com o tipo da variável, e este tipo pode mudar durante a execução). No entanto, quando WebAssembly e JS interagem entre si, um valor inesperado pode ser enviado do JS para WebAssembly. Um exemplo seria a criação de um valor para uma variável que está fora do intervalo de representação no WebAssembly, possivelmente resultando em um *overflow* de inteiro. Isso poderia ser explorado para realizar um *buffer overflow* [McFadden et al. 2018].
- *Stack Overflow:* um exemplo deste ataque no WebAssembly pode ser alcançado no caso de recursão excessiva ou violação nas suposições internas da pilha. Em vez de uma falha, os invasores poderão sobrescrever os dados, uma vez que a proteção na pilha não gerenciada não existe no WebAssembly [Lehmann et al. 2020].
- *Heap Metadata Corruption:* o invasor explora o alocador de memória que foi usado no binário Wasm, uma vez que esta informação é armazenada com o binário [Lehmann et al. 2020].

Compilar um programa de uma determinada linguagem para um binário Wasm é conhecido como portar um programa. Considerando uma perspectiva de segurança, alguns questionamentos aparecem na propagação de vulnerabilidades da linguagem fonte e no surgimento de um comportamento inesperado. Isso ocorre devido a diferentes opções de *design* entre eles. [Stiévenart et al. 2022b] apresenta descobertas sobre programas que travariam na linguagem de origem, mas seriam executados em WebAssembly e diferentes tipos para variáveis.

[Romano et al. 2021] apresenta um estudo sobre *bugs* em compiladores WebAssembly. Como as aplicações WebAssembly tendem a depender de um compilador que converte código de outra linguagem, lidar com os *bugs* torna-se um problema ainda mais complexo. Um total de nove desafios exclusivos para compiladores WebAssembly foram identificados ao considerar o desenvolvimento para WebAssembly. Eles são *Asyncify Synchronous Code*, *Incompatible Data Types*, *Memory Model Differences*, *Bugs in Other Infrastructures*, *Emulating Native Environment*, *Supporting Web APIs*, *Cross-Language Optimizations*, *Runtime Implementation Discrepancy*, e *Unsupported Primitives*. Esses

desafios foram encontrados após investigar o compilador Emscripten, e a maioria dos *bugs* está relacionada à manipulação de *strings* e arquivos.

No geral, ao considerar o compilador e a portabilidade da aplicação, são possíveis três vetores de ataque no WebAssembly. A primeira seria explorar vulnerabilidades em linguagens que serão compiladas no Wasm. Essa estratégia permite que invasores desencadeiem um comportamento inesperado[5] e permite acesso a recursos inacessíveis no WebAssembly. O segundo vetor são as vulnerabilidades encontradas no compilador, essas vulnerabilidades são devidas a recursos particulares encontrados em cada implementação. O terceiro vetor são as vulnerabilidades encontradas no WebAssembly, que existem devido ao seu *design*. Esses vetores poderiam ser combinados para efetuar um ataque e, conforme discutido, uma série de vulnerabilidades e falhas já estão mapeadas.

6.6.3.4. *Overwriting Data*

Overwriting Data é uma primitiva de ataque que permite que invasores sobrescrevam dados de forma que controle adicional seja concedido ao ator mal-intencionado durante a execução do aplicativo [Lehmann et al. 2020]. Algumas das vulnerabilidades usadas para atingir essa capacidade de sobrescrever dados no WebAssembly são:

- *Format String:* se um invasor controlar o formato de uma função `print`, ele poderá ler ou escrever na memória linear. Isso acontece pela falta de suporte a alguns formatos, o que impacta diretamente na operação da função [McFadden et al. 2018].
- *Stack-Based Buffer Overflows:* um *runtime* WebAssembly possui proteções para bloquear aplicativos que escrevem fora dos limites alocados para a memória linear [McFadden et al. 2018]. No entanto, não possui proteção contra a substituição de variáveis encontradas dentro desses limites. As variáveis podem ser substituídas pelo uso de uma função insegura [Lehmann et al. 2020].
- *Cross Site Scripting (XSS):* como as variáveis armazenadas na memória linear podem ser substituídas, as variáveis estáticas não são armazenadas com segurança. As informações poderiam ser escritas sobre posições já ocupadas na memória linear, permitindo um ataque XSS. A memória não é o único vetor para XSS; os invasores também podem controlar ponteiros de função [McFadden et al. 2018].
- *Overwriting Stack Data:* como nenhum mecanismo de segurança está presente em uma pilha não gerenciada, o invasor pode explorar a memória linear sobrescrevendo os dados da função. Porém, o alcance deste ataque é limitado, uma vez que os endereços de retorno não estão presentes na pilha e apenas as chamadas ativas serão exploradas [Lehmann et al. 2020].
- *Overwriting Heap Data:* o *design* de memória no WebAssembly coloca a *heap* após a pilha, sem nenhum mecanismo de defesa para evitar ataques. A corrupção do *heap* é apenas uma questão de estourar a pilha [Lehmann et al. 2020].

[5]Consistem em comportamentos que podem ser desencadeados devido à portabilidade do aplicativo ou interações com outros recursos no ambiente [Lehmann et al. 2020].

- *Overwriting Constant Data:* o *design* da memória linear torna impossível definir uma variável imutável. Um dado constante pode ser substituído ou alterado por um *stack-overflow* ou *stack-based buffer overflow* [Lehmann et al. 2020].

- *Redirect Indirect Calls:* através da substituição da memória linear, um invasor pode alterar um índice que poderia ser usado para emitir uma chamada indireta de função [Lehmann et al. 2020].

- *Code Injection:* permite que invasores façam alterações indesejadas no ambiente. Um exemplo seria no *host*, onde o código pode ser adicionado para substituir argumentos encontrados no aplicativo WebAssembly.

6.6.3.5. *Side-channel Attacks*

Side-channel attacks exploram erros de implementação que permitem que invasores acessem e extraiam informações de um aplicativo. Diferentes estratégias são exploradas de acordo com a aplicação e o ambiente [Spreitzer et al. 2017, Genkin et al. 2018]. A invocação de funções presentes no Emscripten poderia permitir a execução de código no lado do servidor [McFadden et al. 2018].

Uma demonstração de ataques *side-channel* no WebAssembly é apresentada por [Genkin et al. 2018]. O ataque não depende de vulnerabilidade no navegador, explorando o acesso à memória de fora da *sandbox*. O ataque foi feito contra bibliotecas criptográficas através de um código portátil.

Ataques de *timing side-channel* é o foco de [Mazaheri et al. 2022]. Apesar do estudo focar em JS, são apresentadas contramedidas para esses tipos de ataques em JS e WebAssembly. O ataque de *cache* e o ataque *Spectre* são as duas estratégias usadas para realizar o ataque de *side-channel*. As contramedidas são apresentadas com uma abordagem de detecção chamada *Lurking Eyes*, que aplica correções nos níveis de *hardware* e *software*.

6.6.3.6. Classificação

A Tabela 6.1 apresenta uma visão geral dos vetores de ataque discutidos anteriormente, com a respectiva categoria de ataque, a referência onde esta ameaça foi apresentada e referências que propõem contramedidas. As soluções de contramedidas são poucas e têm limitações na hora de serem aplicadas. Para os invasores, o WebAssembly oferece, além de diversas vulnerabilidades, flexibilidade na utilização do *design* do formato WebAssembly para realizar ataques.

6.7. Estratégias de mitigação, monitoramento e detecção de ameaças

Na literatura encontramos tanto propostas que visam melhorar a segurança do ambiente WebAssembly (Seção 6.7.1), como recursos para realizar a análise e detecção de ameaças no ambiente (Seção 6.7.2). Esta seção visa contextualizar a respeito destes tópicos.

Tabela 6.1. Visão geral das falhas e vulnerabilidades do WebAssembly usadas em ataques

Categoria	Ameaças	Contramedidas
Buffer Overflows		[Michael et al. 2023]
Cross Site Scripting (XSS)	[McFadden et al. 2018, Lehmann et al. 2020]	
Format String	[McFadden et al. 2018]	
Heap Metadata Corruption	[Lehmann et al. 2020]	
Integer Overflow/Underflow	[McFadden et al. 2018]	[Michael et al. 2023]
Stack Based Buffer Overflows	[McFadden et al. 2018, Lehmann et al. 2020]	
Stack Overflow	[Lehmann et al. 2020]	
Overwriting Constant Data	[Lehmann et al. 2020]	
Overwriting Heap Data	[Lehmann et al. 2020]	
Overwriting Stack Data	[Lehmann et al. 2020]	
Server-side Remote Code Execution	[McFadden et al. 2018]	
Side-Channel	[Genkin et al. 2018]	[Narayan et al. 2021, Protzenko et al. 2019]
Redirect Indirect Calls	[Lehmann et al. 2020]	
Code Injection	[Lehmann et al. 2020]	
Constant-Time		[Mazaheri et al. 2022, Tsoupidi et al. 2021, Watt et al. 2019a, Renner et al. 2018]

6.7.1. Melhorando a segurança do WebAssembly

As melhorias na segurança do WebAssembly abrangem estudos que introduzem correções para falhas anteriores de *design* e/ou que adicionam recursos de segurança ao formato e seu ambiente.

[Bosamiya et al. 2022] explora duas técnicas de *sandbox* para WebAssembly, avaliando a segurança e o desempenho das propostas. A primeira técnica usa provas verificadas por máquina para garantir que todas as entradas sejam seguras para serem executadas em uma *sandbox*. A segunda técnica utiliza garantias de segurança comprováveis, aproveitando as salvaguardas presentes no *design* do Rust para trazer segurança e desempenho ao compilador. O tempo de execução não tem impacto no desempenho e a segurança foi aprimorada.

RLBox é um *framework* para *sandbox* de bibliotecas de terceiros que podem oferecer algum perigo à segurança do navegador [Narayan et al. 2020]. Foram avaliados dois mecanismos de isolamento: SFI e baseado em processos multi-core. O impacto da solução no desempenho é modesto, sendo o valor mais alto de 13% na latência e 25% no uso de memória.

Spectre é um ataque que pode contornar as medidas de segurança do WebAssembly [Kocher et al. 2020]. *Swivel* é um *framework* desenvolvido para proteger contra esse tipo de ataque, fortalecendo a segurança da *sandbox* WebAssembly contra ataques de *breakout* e *poisoning* [Narayan et al. 2021]. As propostas têm uma sobrecarga, porém oferecem um ambiente *sandbox* mais seguro usando um isolamento de memória mais

forte.

Alguns autores exploraram o uso de ambientes confiáveis de execução, como Intel SGX [Will et al. 2021], para melhorar a segurança do ambiente de execução WebAssembly:

- *AccTEE* é um *framework sandbox* bidirecional que oferece segurança em casos de computação remota [Goltzsche et al. 2019]. A proteção existe com o uso de Software Guard Extensions (SGX) e *sandbox* WebAssembly para proteger a execução e os dados no ambiente. Além de garantir integridade e confidencialidade, o modelo de solução oferece proteção contra códigos maliciosos e não confiáveis.
- TWINE é um *design* de suporte de tempo de execução que explora SGX e a interface WASI para criar um ambiente confiável para executar aplicativos não modificados que já foram compilados para WebAssembly [Ménétrey et al. 2021]. A proposta é um Trusted Execution Environment (TEE) que utiliza o *sandbox* WebAssembly para abstrair o ambiente da aplicação. Esta proposta traz um ambiente de execução mais seguro para *cloud computing*, além de apresentar uma comparação detalhada de desempenho.
- *Se-Lambda* é uma estrutura de computação *serverless* que se concentra na segurança no ambiente de *cloud computing* [Qiang et al. 2018]. A solução *sandbox* bidirecional protege o *gateway* API, os dados do usuário e a plataforma na Cloud. O *framework* apresenta baixo impacto no desempenho e, como o protótipo é baseado em um projeto de código aberto, novas funcionalidades poderiam ser adicionadas acompanhando novas melhorias no WebAssembly.
- [Pop et al. 2022] apresenta um *design* de enclave baseado em WebAssembly. A proposta garante integridade e confidencialidade, oferecendo um canal seguro para os serviços Wasm migrarem entre diferentes arquiteturas.
- *WaTZ* é um *runtime* WebAssembly baseado em TEE que também oferece suporte a atestação remota [Ménétrey et al. 2022]. O isolamento da *sandbox* WebAssembly evita ataques de elevação de privilégios, com algum impacto no desempenho.
- *Edgedancer* é uma plataforma para *edge computing* móvel que utiliza WebAssembly para garantir segurança e permite a automigração dentro da infraestrutura [Nieke et al. 2021]. A plataforma também aproveita o TEE para melhorar a segurança. Pequenos serviços apresentam melhor desempenho durante a migração com maior padrão de segurança.

Fuzzing é uma técnica de teste de *software* que visa encontrar pontos fracos, falhas e vulnerabilidades de *design* por meio da geração de testes de entrada [Liang et al. 2018]. Esta estratégia permite monitorar e estudar o comportamento de uma aplicação:

- *WAFL* é uma ferramenta de *fuzzing* para binários WebAssembly que usa AFL++, um *fuzzer* bem conhecido [Haßler and Maier 2021]. A solução conecta um aplicativo WebAssembly que está sendo testado com o AFL++. A solução não prejudica o desempenho dos compiladores.

- *Fuzzm* é um *fuzzer* para formato WebAssembly, focado principalmente na detecção de vulnerabilidades na memória [Lehmann et al. 2021]. Através da implementação de canários, o *fuzzer* usa entradas do AFL para detectar *overflows* e *underflows* de memória que podem ser explorados na pilha e no *heap*. O *fuzzer* requer apenas o binário.

- [Chen et al. 2022] apresenta o *WASAI*, um *fuzzer* para identificação de vulnerabilidades em contratos inteligentes implementados em WebAssembly. O estudo constatou que, em um conjunto de 991 amostras, mais de 70% apresentavam algum tipo de vulnerabilidade.

A análise *taint* rastreia o fluxo de entrada de dados não confiáveis através de um aplicativo durante a execução [Ming et al. 2016]. Esta estratégia é importante para verificar violações de políticas e operações inseguras.

- O rastreamento *taint* de aplicações Wasm é apresentado em [Szanto et al. 2018]. Através de uma máquina virtual WebAssembly implementada em JS, o *framework* permite uma estratégia de rastreamento de *taint* para o estudo da sensibilidade dos dados do *bytecode*. A avaliação descreve uma sobrecarga de memória aceitável, com a estrutura sendo limitada apenas por algumas funcionalidades ausentes apresentadas no WebAssembly.

- *TaintAssembly* é um *framework* para rastreamento de *taint* em WebAssembly, permitindo o estudo das interações entre WebAssembly e JS [Fu et al. 2018]. A estrutura tem sobrecarga de desempenho, sofre de limitações de *design* que exigem geração de números aleatórios e não implementa operações de comparação.

SELWasm é uma estrutura para proteger a propriedade intelectual do código WebAssembly [Sun et al. 2019]. A estrutura protege contra a reutilização de código-fonte sem autorização, usando criptografia para garantir que os invasores não consigam obter o código-fonte e uma estratégia de *lazy loading* para otimização.

BLADE é um *framework* para proteção contra vazamentos de especulação de algoritmos criptográficos [Vassena et al. 2021]. O *framework* é implementado no compilador WebAssembly e permite a avaliação de sua implementação através de implementações vulneráveis do WebAssembly. A solução não requer supervisão e pode ser eficaz em outros linguagens.

Com o objetivo de entender as vulnerabilidades por trás da tecnologia WebAssembly para aplicações multiplataforma, o trabalho [André et al. 2022] foca nas dúvidas dos desenvolvedores encontradas no StackOverflow[6] para entender melhor os problemas de segurança. Mais da metade das perguntas são direcionadas a informações sobre recursos ou *bugs*. A autenticação é o grupo de questões mais frequentes.

Considerando o impacto na segurança da compilação de aplicações de C para WebAssembly, e explorando que tipo de comportamento poderia ser esperado, os autores

[6]StackOverflow é um site focado em perguntas e respostas [stackoverflow 2023].

em [Stiévenart et al. 2021] apresentam um estudo com foco na avaliação das vulnerabilidades/divergências que são portadas de C após o processo de compilação ser concluído. Em um conjunto de dados com 4,4 mil amostras, 24% dos aplicativos testados tiveram um resultado diferente, devido à falta de medidas de segurança como canários de pilha quando portados para WebAssembly.

6.7.2. Análise e proteções para WebAssembly

A análise de aplicações WebAssembly permite que os desenvolvedores tenham um melhor entendimento do ambiente em que as aplicações serão executadas, oferecendo informações sobre interações com outras linguagens, impactos no desempenho e segurança. Medidas de proteção poderiam ser tomadas após este tipo de estudo.

[Romano and Wang 2020a] apresenta *WASim*, uma ferramenta para classificação de módulos WebAssembly. Onze categorias foram usadas para treinar quatro modelos de aprendizado de máquina. O conjunto de dados foi extraído do Alexa 1 milhão, com os modelos atingindo uma precisão de 91,6%.

[Helpa et al. 2023, Helpa et al. 2024] apresenta uma estratégia para a detecção de anomalias em binários WebAssembly. A proposta extrai características dos binários através do uso do formato Debugging With Attributed Record Format (DWARF), sendo um formato para depuração que permite acesso a seções específicas do binário.

[Heinrich et al. 2023, Heinrich et al. 2024] apresentam duas estratégias de detecção de anomalias para aplicações WebAssembly. As estratégias coletam chamadas WASI realizadas pelo ambiente *sandbox* do WebAssembly, posteriormente utilizando estes dados em uma estratégia de categorização que permite a detecção de anomalias das aplicações.

WebAssembly Symbolic Processor (WASP) é o foco em [Marques et al. 2022]. Essas estratégias permitem que os desenvolvedores encontrem *bugs* e falhas de segurança por meio da análise de múltiplos caminhos de programas.

Oron é uma plataforma de instrumentação que foi implementada usando WebAssembly para resolver problemas de desempenho do *Linvail* [Munsters et al. 2021]. A plataforma foi avaliada através da instrumentação do código AssemblyScript. As avaliações mostram menos sobrecarga de desempenho em comparação com outras soluções.

Wasmati é uma ferramenta de análise estática apresentada por [Brito et al. 2022], com foco em encontrar vulnerabilidades em binários Wasm. *Wasmati* foi desenvolvido para ser utilizado na fase de desenvolvimento, limitando-se a binários baseados em Emscripten. A técnica utilizada para a avaliação consiste em Code Property Graph (CPG) que inclui informações sobre o código que está sendo analisado como pares propriedade-valor.

[Lehmann and Pradel 2019] apresenta *Wasabi*, que é uma estrutura para avaliação dinâmica de WebAssembly. A primeira contribuição consiste em um levantamento de abordagens com análise dinâmica encontradas na literatura. *Wasabi* aproveita a implementação do WebAssembly para extrair informações em tempo de execução, além de não se limitar apenas a aplicações implementadas em WebAssembly.

[Stiévenart and De Roover 2020] analisa binários Wasm por meio do isolamento

de segmentos do código. A avaliação se concentra no desenvolvimento de uma análise de fluxo das aplicações WebAssembly. Apesar de uma precisão baixa de 64%, a contribuição do artigo consiste em uma análise estática que permite a avaliação de funcionalidades do código WebAssembly.

EOSafe é uma estrutura que analisa o binário Wasm para encontrar vulnerabilidades de contratos inteligentes EOSIO[7] [He et al. 2021]. A estratégia de detecção depende de heurísticas e da limitação da execução simbólica. Quatro das vulnerabilidades mais populares para EOSIO são exploradas, com os modelos alcançando um *f1score* de 98%.

WasmView é um *framework* para avaliar/visualizar a interação entre JS e WebAssembly [Romano and Wang 2020b]. O *framework* permite o entendimento de chamadas de função entre as linguagens e *stack traces* das aplicações, através da apresentação de um gráfico visual de chamadas e *trace logs* das informações capturadas.

6.8. Tendências recentes e problemas abertos de pesquisa

Para compreender as tendências e problemas em aberto ao considerar o formato WebAssembly, são consideradas seis das principais características na literatura, estas sendo as novas propostas para aperfeiçoar o formato (Seção 6.8.1), tipos de aplicações em que o WebAssembly está sendo adotado (Seção 6.8.2), conjuntos de dados que estão disponíveis e podem ser utilizados para o estudo do WebAssembly (Seção 6.8.3), as limitações encontradas (Seção 6.8.4), os tópicos de pesquisa em aberto (Seção 6.8.5), e o estado atual de desenvolvimento (Seção 6.8.6).

6.8.1. Novas propostas para o formato WebAssembly

Conforme discutido na Seção 6.6.3, apesar do foco na segurança por trás do *design* do WebAssembly, falhas e vulnerabilidades existem e podem ser exploradas por usuários mal intencionados. Porém, existe um conjunto de trabalhos que visam propor melhorias de *design* para o formato WebAssembly, apresentando contramedidas e propostas para aumentar a segurança no ambiente. Esta seção tem como objetivo discutir esses trabalhos.

Constant-Time WebAssembly (CT-Wasm) é uma expansão do compilador WebAssembly, com foco principal em segurança criptográfica [Watt et al. 2019a]. CT-Wasm permite a verificação de propriedades de segurança e é seguro contra ataques de *side-channel*. A proteção de dados permite que os desenvolvedores melhorem a segurança considerando o fluxo de informações. [Renner et al. 2018] também foca na limitação de segurança do tempo constante no WebAssembly para primitivas criptográficas. Para atingir este objetivo, o compilador WebAssembly foi modificado, adicionando-se o suporte à sensibilidade de um tipo de variável. A ampliação do processo de validação com adição de funcionalidades de verificador de tipo, e o ambiente de execução foi modificado para garantir a segurança. A verificação de primitivas criptográficas no WebAssembly é o foco do [Protzenko et al. 2019]. Um conjunto de ferramentas para compilar F*[8] para WebAssembly é implementada, o que oferece a opção de usar bibliotecas de plataforma via WebAssembly. O processo de validação consiste em compilar HACL*[9] para WebAs-

[7]EOSIO é um protocolo blockchain [EOSIO 2023].

[8]F* é uma linguagem de programação de uso geral orientada para provas [fstar-lang 2023].

[9]HACL* é uma biblioteca criptográfica escrita em F* [HACL* 2023].

sembly e verificar a proteção contra ataques de *side-channel*.

Gobi é um SFI para WebAssembly, que visa resolver as limitações do *sandboxing* do WebAssembly [Narayan et al. 2019]. O sistema agrupa compiladores e alterações no tempo de execução que permitem que bibliotecas implementadas em linguagens como C/C++ sejam compiladas para WebAssembly. Este protótipo do WebAssembly SFI foi incorporado pela comunidade Wasm através da criação do WebAssembly System Interface (WASI) e melhorias no conjunto de ferramentas Lucet (no entanto, em 2020 o foco foi alterado para utilizar o Wasmtime [BytecodeAlliance 2023]) .

A contribuição de [Watt et al. 2019b] está diretamente ligada ao desenvolvimento do padrão Wasm, apresentando extensões de memória e operação para WebAssembly e JS. As extensões WebAssembly permitem o uso de *threads*, atômicos e mutabilidade. O modelo de memória garante o uso sequencial de dados e a simultaneidade de memória compartilhada é garantida para implementações futuras.

O WebAssembly possui problemas relacionados à memória que ainda ocorrem dentro do *sandbox* (Seção 6.6.3). Memory-safe WebAssembly (MSWasm) é uma extensão para segurança de memória apresentada por [Michael et al. 2023]. A especificação formal define o uso de memória segmentada para segurança de memória. Duas implementações são discutidas, a MSWasm para segurança de memória em compiladores WebAssembly e um compilador C para MSWasm que garante segurança de memória contra fontes inseguras. A extensão quando avaliada apresentou um *overhead* de 197,5%.

[Vassena and Patrignani 2020] discute problemas de memória encontrados no WebAssembly e aponta soluções conhecidas que podem ser exploradas para solucionar estas limitações. Através da utilização da extensão MS-Wasm, a proposta pode garantir a segurança da memória.

Uma extensão do WebAssembly é apresentada em [Disselkoen et al. 2019], que apresenta um *design* para garantir a segurança espacial e temporal e a integridade do ponteiro. Com segurança de memória explícita no nível da linguagem, a implementação proposta do WebAssembly garante segurança de memória.

[Kolosick et al. 2022] faz modificações no *sandbox* WebAssembly para garantir melhor desempenho e segurança. O *sandbox* Wasm usa transições que são pesadas durante o tempo de execução, mudar isso para condições de custo zero também atinge o mesmo nível de garantia de segurança com melhor desempenho. Através de um verificador chamado *VeriZero* é possível identificar quando uma função é semanticamente capaz de explorar esta característica.

WebAssembly sofre com consumo de memória e tempo de inicialização, um *design* para uma solução eficaz é o foco em [Titzer 2022]. A proposta utiliza uma tabela lateral compacta gerada durante o processo de validação para fornecer melhor desempenho para o Wasm.

SecWasm é um *design* de sistema Information-Flow Control (IFC) híbrido para a confidencialidade de dados em aplicações WebAssembly [Bastys et al. 2022]. A solução supera problemas de fluxo de controle estruturado e memória linear, mas seu impacto no desempenho é desconhecido, uma vez que ela não foi implementada.

[Arteaga 2022] apresenta solução para mitigar a monocultura de *software* em WebAssembly. Randomização e execução multivariável foram propostas para os *frameworks* Code Randomization of WebAssembly (CROW) e Multi-variant Execution for WebAssembly (MEWE).

6.8.2. Aplicações variadas do formato WebAssembly

O uso do WebAssembly para resolver problemas encontrados em diversos cenários é considerado nesta seção. Como o WebAssembly oferece uma gama de recursos, diferentes aplicações podem explorar essa tecnologia para obter desempenho e ganho de segurança.

Na literatura, um conjunto de trabalhos foca na aplicação do WebAssembly para melhorar o ambiente IoT.

- [Radovici et al. 2018] propõe uma estrutura que se concentra na execução de *bytecodes* em uma *sandbox* WebAssembly isolada. A proposta depende principalmente do ambiente WebAssembly, com foco em um *design* para expor com segurança os recursos necessários para executar uma aplicação. Espera-se que a proposta contribua para a segurança e desempenho no cenário IoT.

- *Aerogel* é uma estrutura para proteger melhor o controle de acesso entre aplicações *bare-metal* e IoT usando o ambiente WebAssembly, abordando lacunas de segurança anteriormente não abordadas [Liu et al. 2021]. A estrutura utiliza o *sandbox* WebAssembly para aplicar medidas de controle de acesso, com uma sobrecarga de desempenho de 1% e o crescimento do consumo de energia atingindo quase 46%.

- *Wasmachine* é um sistema operacional (SO) focado principalmente na segurança do WebAssembly para a execução de aplicações em IoT e dispositivos de *fog computing* [Wen and Weber 2020]. Através da execução do binário WebAssembly no *kernel mode*, o custo de execução é reduzido. O artigo também implementa o *kernel* do sistema operacional em Rust para garantir a segurança da memória.

- *ThingSpire* é um SO de *cloud-edge* baseado em WebAssembly [Li et al. 2021]. A proposta aborda três desafios relacionados ao desenvolvimento da infraestrutura; (i) como projetar o ambiente de forma eficaz; (ii) como ativar a intercomunicação entre módulos; e (iii) como a segurança e a tolerância a falhas são suportadas.

- *MEWE* é um *framework* que apresenta uma técnica para *edge-cloud computing* [Arteaga et al. 2022]. A estrutura adiciona randomização para execução do tempo de execução do WebAssembly e diversifica os binários do Wasm. A combinação dessas duas estratégias fortalece ataques como Break-Once-Break-Everywhere (BOBE). Com dados reais, foi possível identificar flexibilidade para gerar variantes binárias.

Como *edge computing* possui uma demanda por desempenho, [Koren 2021] apresenta uma prova de conceito do uso de WebAssembly para microcontroladores. Um servidor web de um Integrated Development Environment (IDE) foi desenvolvido para entregar novos módulos para dispositivos *edge computing* e *cloud*.

WebCloud é uma solução de criptografia para comunicação entre serviços em Cloud e navegadores [Sun et al. 2022]. A aplicação *client-side* resolve limitações relacionadas à segurança e controle de acesso de soluções anteriores.

[Baumgärtner et al. 2019] propõe uma aplicação de WebAssembly no contexto de Disruption-Tolerant Networking (DTN). A rede é estabelecida por meio de uma combinação de Bundle Protocol 7 e WebAssembly. Uma aplicação web habilitada pelo navegador realiza a troca de mensagens.

Sledge é um *framework serverless* que utiliza WebAssembly para *edge computing* [Gadepalli et al. 2020]. Oferece isolamento leve de funções em tempo de execução implementado em WebAssembly e políticas de agendamento para a infraestrutura. Os resultados mostraram baixa latência e gerenciamento eficiente de simultaneidade.

WasmAndroid é um *runtime* multiplataforma para linguagens nativas no Android que requer apenas a compilação do código para WebAssembly [Wen et al. 2021]. Experimentos com o *benchmark* SPEC CPU 2006 mostraram que o WasmAndroid teve tempo de execução 30% superior ao da execução nativa.

WasmTree é uma implementação de Resource Description Framework (RDF) que explora WebAssembly e Rust para otimizar e ganhar desempenho no ambiente Web [Bruyat et al. 2021]. O uso do WebAssembly não apresentou ganho de desempenho. Porém, a avaliação com a consulta SPARQL apresentou melhoria de desempenho.

[Stiévenart et al. 2022a] propõe uma estratégia estática de *slicing* para programas WebAssembly. O estudo visa uma abordagem estática de *intra-procedural backward slicing* para WebAssembly. A proposta é avaliada com aplicações reais.

[Ménétrey et al. 2022] apresenta um posicionamento para o desenvolvimento de aplicações que possam rodar em um conjunto de dispositivos de *hardware* sem perda de desempenho e segurança. O estudo tem como foco descrever o impacto que o uso do WebAssembly pode trazer para esse meio.

Uma avaliação do uso de Ethereum Virtual Machine (EVM) e WebAssembly para contratos inteligentes é o foco em [Zhang et al. 2024]. O uso do WebAssembly nos clientes blockchain Ethereum apresenta problemas relacionados ao suporte, limitações de desenvolvimento e instabilidade. O desempenho do WebAssembly varia significativamente.

Hector é uma estrutura de aplicação web que permite aplicações distribuídas no navegador web [Goltzsche et al. 2020]. WebAssembly é um dos componentes que garantem integridade, confidencialidade e isolamento.

EVulHunter é uma ferramenta desenvolvida para detecção de vulnerabilidades do EOSIO WebAssembly, focada em ataques de transferência falsa [Quan et al. 2019]. Duas estratégias são utilizadas para a análise estática, funções predefinidas e operações de comparação.

EOSDFA é um *framework* para análise dos contratos inteligentes na blockchain EOSIO [Li and Zhang 2022]. O *framework* expande o sistema de implantação automática conhecido como *framework* Octopus, permitindo a avaliação de *pointer access* para fluxo de controle e análise de fluxo de dados.

ZAWA [Gao et al. 2022] é uma máquina virtual que emula *bytecodes* WebAssembly. Visando um tempo de execução mais seguro, o ZAWA aproveita o Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (ZKSNARK), para avaliar se algum vazamento está acontecendo.

6.8.3. Conjuntos de dados disponíveis

A disponibilidade de dados para experimentação ajuda os pesquisadores a comparar resultados, melhorar pesquisas publicadas anteriormente e desenvolver novas abordagens. A Tabela 6.2 apresenta alguns dos principais conjuntos de dados disponíveis atualmente que podem ser utilizados para o desenvolvimento de pesquisas com WebAssembly, descrevendo o tipo de dados, número de amostras, metodologia de extração e ano de criação.

Tabela 6.2. Datasets WebAssembly

Dataset	Propósito	Amostras	Extração de Dados	Ano
WasmBench [Hilbig et al. 2021]	Uma representação de conjunto de dados do mundo real	8.4 K	Diversas fontes, como: GitHub, gerenciador de pacotes, sites e manualmente	2021
SnowWhite [Lehmann and Pradel 2022]	Recuperação de tipos WebAssembly	6.3 M	4 k amostras de pacotes de código-fonte do Ubuntu C e C++	2022
[Chen et al. 2022]	Identificação de vulnerabilidades de contratos inteligentes Wasm	3.3 k	Vulnerabilidades e contratos inteligentes	2022
QRS [Stiévenart et al. 2021]	Aplicações C com comportamento diferente quando compilados para WebAssembly	1,088	Código C vulnerável do Juliet Test Suite 1.3 (2017)	2021
SAC [Stiévenart et al. 2022b]	Apresenta binários Wasm com problemas de segurança ao portar WebAssembly de C	4.9 K	Código C vulnerável do Juliet Test Suite 1.3 (2017)	2022
Alexa [Amazon 2023]	Uma classificação de 1 milhão dos domínios de maior tráfego em todo o mundo, atualizada diariamente	Variável	Os dados de tráfego são classificados por domínio. Os dados são coletados por meio de uma extensão do navegador usada pelos usuários	2008-2022
Baseado no Alexa [Musch et al. 2019]	Define o uso do WebAssembly na web	150	O módulo WebAssembly de páginas da web foi coletado, com 1.950 amostras (150 exclusivas) de 1.639 sites	2019
Dataset de perguntas [André et al. 2022]	Perguntas relacionadas à segurança do WebAssembly feitas por desenvolvedores	359	Perguntas de segurança do Stack Overflow	2022
Dataset de bugs [Romano et al. 2021]	Coleção de bugs do Emscripten	1,054	Bugs foram extraídos do projeto Emscripten	2021

No geral, apesar da quantidade limitada de conjuntos de dados que podem ser uti-

lizados em estudos de segurança, as amostras mais recentes pretendem representar o atual ecossistema Wasm. É importante notar que estes conjuntos de dados não estão sendo atualizados com amostras mais recentes. Isto pode ser considerado arriscado, considerando que o WebAssembly ainda está passando por grandes mudanças e que dados mais recentes e mais representativos serão necessários no futuro.

O ranking Alexa [Amazon 2023] é uma fonte recorrente de dados para experimentos não apenas em WebAssembly. Na coleta de dados apresentada por [Kim et al. 2022], apenas uma abordagem não explora o ranking Alexa[10]. No entanto, o uso do ranking Alexa para pesquisas de segurança não é adequado, uma vez que o método utilizado para coleta de dados varia muito e a falta de detalhes pode afetar os resultados da pesquisa [Le Pochat et al. 2019, Ruth et al. 2022]. Algumas alternativas mais adequadas para representar o ambiente Web para pesquisas de segurança são discutidas em [Xie et al. 2022, Le Pochat et al. 2019, Durumeric 2023].

Também é importante considerar o tipo de informação que cada conjunto de dados fornece. Para a extração de informações de binários, pode-se utilizar a maioria das amostras disponíveis nos conjuntos de dados apresentados. Porém, para uma avaliação que requer a execução dos binários, restrições relacionadas à falta de dependências do período de extração de dados ou mesmo suporte dos dados, limitam o uso desse conjunto de dados. Uma alternativa, neste caso, seria usar ferramentas de *benchmark* e conjuntos de testes que funcionam em compiladores.

6.8.4. Limitações atuais do WebAssembly

Apesar do foco na segurança, algumas opções de *design* no WebAssembly limitam as medidas de segurança encontradas em outras linguagens que podem ser compiladas para o WebAssembly. Em alguns casos, esses mecanismos não estarão presentes quando o código for compilado de outra linguagem para Wasm [McFadden et al. 2018].

Address Space Layout Randomization (ASLR) é uma proteção de memória que, apesar de estar presente em outras linguagens, seria difícil de ser implementada em WebAssembly, considerando o *design* da memória linear [McFadden et al. 2018]. No futuro, esta proteção poderá ser adicionada [W3C Community Group 2022a]. No entanto, a memória linear também elimina a necessidade de alguma proteção como canários de pilha. O *design* do WebAssembly também remove os requisitos de recursos como Data Execution Prevention (DEP), uma vez que instruções de baixo nível não terão o mesmo acesso aos recursos, portanto esta proteção não é necessária.

O *Heap Hardening* é apresentado em alguns compiladores e soluções foram propostas na literatura. Além disso, os compiladores que suportam Control Flow Integrity (CFI) podem exportar a proteção para o Wasm compilado [McFadden et al. 2018].

6.8.5. Tópicos de pesquisa em aberto

As discussões sobre vulnerabilidades do WebAssembly estão evoluindo e é possível perceber mudanças feitas pela comunidade com base nessas descobertas. Contudo, na prá-

[10]O projeto do rank da Alexa foi descontinuado em 2022, uma alternativa acaba sendo o Tranco [Le Pochat et al. 2019].

tica, o uso malicioso do WebAssembly tem sido limitado até agora. Vimos estudos sobre como as interações entre o WebAssembly e os recursos apresentados no ambiente podem ser úteis para invasores e o impacto do formato binário na ofuscação. Estudos futuros devem estar mais alinhados na avaliação do uso atual do WebAssembly por invasores, na eficácia do *sandbox* WebAssembly e nas reais vantagens em áreas específicas como IoT, Trusted Computing (TC) e *blockchains*, nas quais se pode observar a ampla aplicabilidade do WebAssembly.

Os estudos que visam descrever a adoção e o uso de aplicações WebAssembly no mundo real ainda são limitados, muitos deles não representativos do estado atual da Internet. A discussão sobre a adoção do WebAssembly, mesmo considerando o ponto de vista da segurança, também apresenta os mesmos problemas. Estes problemas estão diretamente associados (i) à utilização de bases de dados desatualizadas, que em alguns casos já se revelaram não representativas; e (ii) à falta de amostras, estratégias de validação e preocupações com a reprodutibilidade dos dados utilizados em estudos futuros. A disponibilidade é considerada, porém, a utilidade dos dados para estudos futuros não é considerada.

Atestados de medidas de segurança no formato WebAssembly não são estudados. A discussão sobre segurança na literatura poderia ser ampliada para o projeto/desenvolvimento do WebAssembly, visando um melhor entendimento da segurança do formato. Uma discussão semelhante seria útil para esclarecer o impacto da migração de algumas linguagens para o WebAssembly, onde várias discussões sobre segurança estão se concentrando.

6.8.6. Estado atual do desenvolvimento do WebAssembly

Os avanços da comunidade nos últimos anos demonstram o potencial que o formato WebAssembly pode alcançar, com constante atualizações e lançamento de novas *features* como o suporte para *tail calls* em 2023. No final de 2023 também foi lançado o suporte para o GC que está disponível no Google Chrome e no Mozilla Firefox. O suporte para o Single Instruction, Multiple Data (SIMD) também apresentado, que pode trazer ainda mais desempenho para aplicações em WebAssembly. Suporte para múltiplas memórias foi lançado em 2023, permitindo que um módulo possua mais de uma memória linear [WebAssembly 2024]. Para 2024 é esperado o lançamento do *snapshot_preview2* da WASI[11].

6.9. Considerações finais

O WebAssembly é um formato *bytecode* que vem para trazer mais segurança e desempenho para o ambiente da Web. O formato permite que desenvolvedores não só implementem aplicações diretamente utilizando o formato textual WAT, como também possibilita que linguagens de alto nível sejam utilizadas pelos desenvolvedores já que posteriormente estes códigos podem ser portados para o formato WebAssembly.

Através de um ambiente *sandbox*, aplicações desenvolvidas em WebAssembly podem ser executadas em um conjunto variado de arquiteturas, além de oferecer maior

[11] Detalhes das propostas e suas respectivas fases podem ser encontrados em [WebAssembly 2024].

segurança ao executar código de terceiros no *client-side*. A versão atual já consegue demonstrar as vantagens das principais características do modelo, apesar de ainda estar em um processo inicial de desenvolvimento e teste. Claro que limitações existem, mas as vantagens já são evidentes ao considerar maior segurança e desempenho no ambiente da Web.

O suporte para o formato WebAssembly já alcança muitas das principais linguagens, e uma variedade de compiladores estão disponíveis [Stephen 2022]. A adoção do formato também é alta, com os principais navegadores tendo suporte aos principais recursos. As propostas também apresentam o mesmo ritmo, com lançamentos constantes de novas funcionalidades. Wasmtime e Emscripten são dois dos compiladores que possuem um projeto ativo, com novos recursos sendo adicionados com frequência e comunidades ativas.

Na academia, o desempenho e a segurança do WebAssembly vêm sendo estudados de perto pela comunidade, com novos estudos sendo publicados discutindo as principais mudanças e impacto. Estes trabalhos também discutem novas propostas que podem gerar melhorias para o formato WebAssembly. Por exemplo, as medidas de segurança para o WebAssembly não estão limitadas a um problema específico. A melhoria da segurança está sendo alcançada por novas propostas para o *design* do WebAssembly, *frameworks* que melhoram a segurança, ou estratégias de avaliação para binários Wasm.

A melhoria da segurança do ambiente está sendo alcançada por meio de mudanças no *sandbox*, explorando recursos como SGX e API para fortalecer as proteções de tempo de execução do WebAssembly. Para o desenvolvimento foram desenvolvidas estratégias com *fuzzing*, *tracing* e *taint* para demonstrar como os recursos interagem em uma aplicação. Também é possível apontar o desenvolvimento de estratégias de análise e proteção para aplicações WebAssembly.

Novas propostas estão abordando os principais problemas do WebAssembly com melhorias no *design* do formato. Essas melhorias estão sendo aproveitadas pela comunidade para fazer mudanças no formato para garantir um ambiente mais seguro.

O formato também possui limitações e problemáticas que ainda precisam ser abordado pela comunidade. Problemas de segurança existirão em qualquer formato, pois *bugs*/falhas podem ocorrer devido à forma como os recursos e implementações são feitos pelos desenvolvedores. No WebAssembly, os invasores podem explorar vulnerabilidades ou falhas de *design* que facilitam operações maliciosas, apesar da segurança envolvida no *design* do WebAssembly. A maioria das questões de segurança discutidas neste capítulo está relacionada a alguma limitação ou escolha de projeto.

A memória linear é um dos destaques de *design* do WebAssembly, isolando a aplicação, pois os acessos à memória são limitados a uma região específica [Kim et al. 2022]. No entanto, algumas vulnerabilidades de memória persistem devido à falta de salvaguardas de segurança em linguagens que podem ser portadas para Wasm. Não se limitando à memória, um dos recursos mais atraentes do WebAssembly pode ser um dos mais perigosos. A portabilidade de uma variedade de linguagens impacta diretamente a segurança do binário Wasm, uma vez que uma variedade de linguagens implementam diferentes proteções que não estão presentes no WebAssembly e, em muitos casos, podem ser trans-

portadas ao gerar um binário Wasm.

As falhas/vulnerabilidades de segurança discutidas na literatura exploram diversos elementos. No entanto, a maioria desses problemas não se limita ao WebAssembly, portanto, enfatizar esse problema como uma restrição ao uso do WebAssembly não é adequado. Considerando o estado atual, foram feitos avanços para um ambiente mais seguro e foram propostas soluções para corrigir falhas/vulnerabilidades de segurança.

Malfeitores também podem explorar os recursos oferecidos pelo WebAssembly para realizar uma série de ataques. Os principais atrativos do WebAssembly para usuários maliciosos são (i) fornecer um formato binário (diferente de linguagens interpretadas, como JS), que pode ajudar na ofuscação de código; (ii) o ganho de desempenho do WebAssembly em comparação com outras linguagens no mesmo ambiente o torna atrativo para *cryptojacking*; (iii) por ser um recurso suportado pela maioria dos navegadores web e pela proximidade com JS, aplicações WebAssembly maliciosas tornam-se interessantes.

Na literatura também observamos lacunas que oportunizam futuras pesquisas. Essas lacunas incluem (i) a limitação de discussões entre a interação do WebAssembly com outros recursos encontrados no navegador; (ii) a falta de conjuntos de dados reais, que sejam representativos do contexto atual de uso do WebAssembly na Web; (iii) a definição de métricas adequadas de avaliação de desempenho ao considerar que módulos WebAssembly podem ser executados em diferentes arquiteturas, e serem portados de diversas linguagens de programação.

Agradecimentos

Este trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES), UDESC e FAPESC. Os autores também agradecem o Programa de Pós-Graduação em Informática da UFPR e a UTFPR *Campus* Dois Vizinhos.

Referências

[Adobe 2021] Adobe (2021). Adobe flash player eol general information page. `https://www.adobe.com/products/flashplayer/end-of-life.html`.

[Amazon 2023] Amazon (2023). Alexa Rank. `https://www.alexa.com/company`.

[André et al. 2022] André, P. M., Stiévenart, Q., and Ghafari, M. (2022). Developers struggle with authentication in Blazor WebAssembly. In *Proceedings of the 38th International Conference on Software Maintenance and Evolution*, pages 389–393, Limassol, Cyprus. IEEE.

[Apodaca 2020] Apodaca, R. L. (2020). First steps in webassembly: Hello world. Depth-First. `https://depth-first.com/articles/2020/01/13/first-steps-in-webassembly-hello-world/`.

[Arteaga 2022] Arteaga, J. C. (2022). *Artificial Software Diversification for WebAssembly*. PhD thesis, KTH Royal Institute of Technology, Stockholm, Sweden.

[Arteaga et al. 2022] Arteaga, J. C., Laperdrix, P., Monperrus, M., and Baudry, B. (2022). Multi-variant execution at the edge. In *Proceedings of the 9th Workshop on Moving Target Defense*, pages 11–22, Los Angeles, CA, USA. ACM.

[Arteaga et al. 2021] Arteaga, J. C., Malivitsis, O., Perez, O. V., Baudry, B., and Monperrus, M. (2021). CROW: Code diversification for WebAssembly. In *Proceedings of the Workshop on Measurements, Attacks, and Defenses for the Web*, Virtual Event. Internet Society.

[Balakrishnan and Schulze 2005] Balakrishnan, A. and Schulze, C. (2005). Code obfuscation literature survey. *CS701 Construction of Compilers*, 19.

[Bandhakavi et al. 2010] Bandhakavi, S., King, S. T., Madhusudan, P., and Winslett, M. (2010). VEX: Vetting browser extensions for security vulnerabilities. In *Proceedings of the 19th USENIX Security Symposium*, Washington, DC, USA. USENIX Association.

[Barth et al. 2008] Barth, A., Jackson, C., Reis, C., and The Google Chrome Team, . (2008). The security architecture of the Chromium browser. Technical report, Stanford University. `https://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf`.

[Bastys et al. 2022] Bastys, I., Algehed, M., Sjösten, A., and Sabelfeld, A. (2022). SecWasm: Information flow control for WebAssembly. In *Proceedings of the 29th International Static Analysis Symposium*, pages 74–103, Auckland, New Zealand. Springer.

[Battagline 2021] Battagline, R. (2021). *The Art of WebAssembly: Build Secure, Portable, High-Performance Applications*. No Starch Press.

[Baumgärtner et al. 2019] Baumgärtner, L., Höchst, J., and Meuser, T. (2019). B-DTN7: Browser-based disruption-tolerant networking via bundle protocol 7. In *Proceedings of the 6th International Conference on Information and Communication Technologies for Disaster Management*, pages 1–8, Paris, France. IEEE.

[Bhansali et al. 2022] Bhansali, S., Aris, A., Acar, A., Oz, H., and Uluagac, A. S. (2022). A first look at code obfuscation for WebAssembly. In *Proceedings of the 15th Conference on Security and Privacy in Wireless and Mobile Networks*, pages 140–145, San Antonio, TX, USA. ACM.

[Bian et al. 2019] Bian, W., Meng, W., and Wang, Y. (2019). Poster: Detecting WebAssembly-based cryptocurrency mining. In *Proceedings of the 26th Conference on Computer and Communications Security*, pages 2685–2687, London, UK. ACM.

[Bian et al. 2020] Bian, W., Meng, W., and Zhang, M. (2020). Minethrottle: Defending against Wasm in-browser cryptojacking. In *Proceedings of The Web Conference*, pages 3112–3118, Taipei, Taiwan. ACM.

[Bosamiya et al. 2022] Bosamiya, J., Lim, W. S., and Parno, B. (2022). Provably-Safe multilingual software sandboxing using WebAssembly. In *Proceedings of the 31st*

USENIX Security Symposium, pages 1975–1992, Boston, MA, USA. USENIX Association.

[Brito et al. 2022] Brito, T., Lopes, P., Santos, N., and Santos, J. F. (2022). Wasmati: An efficient static vulnerability scanner for WebAssembly. *Computers & Security*, 118:102745.

[Bruyat et al. 2021] Bruyat, J., Champin, P.-A., Médini, L., and Laforest, F. (2021). WasmTree: Web Assembly for the semantic Web. In *Proceedings of the European Semantic Web Conference*, pages 582–597, Virtual Event. Springer.

[BytecodeAlliance 2021a] BytecodeAlliance (2021a). Wasmtime. `https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-overview.md`.

[BytecodeAlliance 2021b] BytecodeAlliance (2021b). Wasmtime. `https://docs.wasmtime.dev/introduction.html`.

[BytecodeAlliance 2023] BytecodeAlliance (2023). Lucet has reached End-of-life. `https://hacl-star.github.io/`.

[BytecodeAlliance 2024] BytecodeAlliance (2024). Wasmtime. `https://github.com/bytecodealliance/wasmtime`.

[Chen et al. 2022] Chen, W., Sun, Z., Wang, H., Luo, X., Cai, H., and Wu, L. (2022). WASAI: Uncovering vulnerabilities in Wasm smart contracts. In *Proceedings of the 31st International Symposium on Software Testing and Analysis*, pages 703–715, Virtual Event. ACM.

[De Macedo et al. 2021] De Macedo, J., Abreu, R., Pereira, R., and Saraiva, J. (2021). On the runtime and energy performance of WebAssembly: Is WebAssembly superior to JavaScript yet? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 255–262. IEEE.

[De Macedo et al. 2022] De Macedo, J., Abreu, R., Pereira, R., and Saraiva, J. (2022). WebAssembly versus JavaScript: Energy and runtime performance. In *Proceedings of the International Conference on ICT for Sustainability*, pages 24–34, Plovdiv, Bulgaria. IEEE.

[Dejaeghere et al. 2023] Dejaeghere, J., Gbadamosi, B., Pulls, T., and Rochet, F. (2023). Comparing security in eBPF and WebAssembly. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*, pages 35–41, New York, NY, USA. ACM.

[Disselkoen et al. 2019] Disselkoen, C., Renner, J., Watt, C., Garfinkel, T., Levy, A., and Stefan, D. (2019). Position paper: Progressive memory safety for WebAssembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–8, Phoenix, AZ, USA. ACM.

[Donovan et al. 2010] Donovan, A., Muth, R., Chen, B., and Sehr, D. (2010). PNaCl: Portable native client executables. White paper, Google. https://www.chromium.org/nativeclient/reference/research-papers/pnacl.pdf.

[Durumeric 2023] Durumeric, Z. (2023). Cached chrome top million websites. https://github.com/zakird/crux-top-lists.

[Emscripten 2021] Emscripten (2021). Emscripten is a complete compiler toolchain to WebAssembly, using LLVM, with a special focus on speed, size, and the Web platform. https://emscripten.org/.

[EOSIO 2023] EOSIO (2023). EOSIO is a next-generation, open-source blockchain protocol with industry-leading transaction speed and flexible utility. https://github.com/EOSIO.

[Feldman 2019] Feldman, R. (2019). Why isn't functional programming the norm? https://tinyurl.com/yyhtxt74.

[Felker 2023] Felker, R. (2023). Musl libc. https://musl.libc.org/.

[Fox and Patterson 2021] Fox, A. and Patterson, D. (2021). *Engineering Software as a Service: An Agile Approach Using Cloud Computing*. Version 2.0b7, 2nd edition. https://saasbook.info/.

[Fraiwan et al. 2012] Fraiwan, M., Al-Salman, R., Khasawneh, N., and Conrad, S. (2012). Analysis and identification of malicious JavaScript code. *Information Security Journal: A Global Perspective*, 21(1):1–11.

[fstar-lang 2023] fstar-lang (2023). Introduction to F*. https://fstar-lang.org/.

[Fu et al. 2018] Fu, W., Lin, R., and Inge, D. (2018). TaintAssembly: Taint-based information flow control tracking for WebAssembly. arXiv preprint 1802.01050. http://doi.org/10.48550/arxiv.1802.01050.

[Gadepalli et al. 2020] Gadepalli, P. K., McBride, S., Peach, G., Cherkasova, L., and Parmer, G. (2020). Sledge: A serverless-first, light-weight Wasm runtime for the edge. In *Proceedings of the 21st International Middleware Conference*, pages 265–279, Delft, Netherlands. ACM.

[Gao et al. 2022] Gao, S., Fu, H., Zhang, H., Zhang, J., and Li, G. (2022). ZAWA: A ZKSNARK WASM emulator. *Proceedings of the ACM on Programming Languages*, 1(1).

[Genkin et al. 2018] Genkin, D., Pachmanov, L., Tromer, E., and Yarom, Y. (2018). Drive-by key-extraction cache attacks from portable code. In *Proceedings of the 16th International Conference on Applied Cryptography and Network Security*, pages 83–102, Leuven, Belgium. Springer.

[Golsch 2019] Golsch, L. (2019). WebAssembly: Basics. Technical report, Technical University of Braunschweig. https://www.lennard-golsch.de/article/webassembly.pdf.

[Goltzsche et al. 2019] Goltzsche, D., Nieke, M., Knauth, T., and Kapitza, R. (2019). AccTEE: A WebAssembly-based two-way sandbox for trusted resource accounting. In *Proceedings of the 20th International Middleware Conference*, pages 123–135, Davis, CA, USA. ACM.

[Goltzsche et al. 2020] Goltzsche, D., Siebels, T., Golsch, L., and Kapitza, R. (2020). Hector: Using untrusted browsers to provision Web applications. arXiv 2010.09512. 10.48550/arXiv.2010.09512.

[Grosskurth and Godfrey 2006] Grosskurth, A. and Godfrey, M. W. (2006). Architecture and evolution of the modern web browser. *Preprint submitted to Elsevier Science*, 12(26):235–246.

[Haas et al. 2017] Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. (2017). Bringing the Web up to speed with WebAssembly. In *Proceedings of the 38th Conference on Programming Language Design and Implementation*, pages 185–200, Barcelona, Spain. ACM.

[HACL* 2023] HACL* (2023). A High Assurance Cryptographic Library. https://hacl-star.github.io/.

[Haßler and Maier 2021] Haßler, K. and Maier, D. (2021). WAFL: Binary-only WebAssembly fuzzing with fast snapshots. In *Proceedings of the 5th Reversing and Offensive-Oriented Trends Symposium*, pages 23–30, Vienna, Austria. ACM.

[He et al. 2021] He, N., Zhang, R., Wang, H., Wu, L., Luo, X., Guo, Y., Yu, T., and Jiang, X. (2021). EOSAFE: Security analysis of EOSIO smart contracts. In *Proceedings of the 30th USENIX Security Symposium*, pages 1271–1288, Vancouver, BC, Canada. USENIX Association.

[Heinrich et al. 2023] Heinrich, T., Will, N. C., Obelheiro, R. R., and Maziero, C. A. (2023). Uso de chamadas WASI para a identificação de ameaças em aplicações WebAssembly. In *Anais do XXIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, Juiz de Fora, MG, Brasil. SBC.

[Heinrich et al. 2024] Heinrich, T., Will, N. C., Obelheiro, R. R., and Maziero, C. A. (2024). A categorical data approach for anomaly detection in WebAssembly applications. In *Proceedings of the 10th International Conference on Information Systems Security and Privacy*, pages 275–284, Rome, Italy. SciTePress.

[Helpa et al. 2024] Helpa, C., Heinrich, T., Botacin, M., Will, N. C., Obelheiro, R. R., and Maziero, C. (2024). The use of the DWARF Debugging Format for the Identification of Potentially Unwanted Applications (PUAs) in WebAssembly Binaries. In *SECRYPT*.

[Helpa et al. 2023] Helpa, C., Heinrich, T., Botacin, M., Will, N. C., Obelheiro, R. R., and Maziero, C. A. (2023). Uma estratégia dinâmica para a detecção de anomalias em binários WebAssembly. In *Anais do XXIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, Juiz de Fora, MG, Brasil. SBC.

[Herman et al. 2014] Herman, D., Wagner, L., and Zakai, A. (2014). asm.js. `http://asmjs.org/spec/latest/`.

[Hilbig et al. 2021] Hilbig, A., Lehmann, D., and Pradel, M. (2021). An empirical study of real-world WebAssembly binaries: Security, languages, use cases. In *Proceedings of The Web Conference*, pages 2696–2708, Ljubljana, Slovenia. ACM.

[Hoffman 2019] Hoffman, K. (2019). *Programming WebAssembly with Rust: unified development for web, mobile, and embedded applications*. The Pragmatic Bookshelf, Raleigh, NC, USA.

[Jangda et al. 2019] Jangda, A., Powers, B., Berger, E. D., and Guha, A. (2019). Not so fast: Analyzing the performance of WebAssembly vs. native code. In *Proceedings of the USENIX Annual Technical Conference*, pages 107–120, Renton, WA, USA. USENIX Association.

[Johnson et al. 2023] Johnson, E., Laufer, E., Zhao, Z., Gohman, D., Narayan, S., Savage, S., Stefan, D., and Brown, F. (2023). WaVe: a verifiably secure WebAssembly sandboxing runtime. In *Proceedings of the Symposium on Security and Privacy*, pages 2940–2955, San Francisco, CA, USA. IEEE.

[Kim et al. 2022] Kim, M., Jang, H., and Shin, Y. (2022). Avengers, Assemble! survey of WebAssembly security solutions. In *Proceedings of the 15th International Conference on Cloud Computing*, pages 543–553, Barcelona, Spain. IEEE.

[Kocher et al. 2020] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al. (2020). Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101.

[Kolosick et al. 2022] Kolosick, M., Narayan, S., Johnson, E., Watt, C., LeMay, M., Garg, D., Jhala, R., and Stefan, D. (2022). Isolation without taxation: Near-zero-cost transitions for WebAssembly and SFI. *Proceedings of the ACM on Programming Languages*, 6(POPL).

[Koren 2021] Koren, I. (2021). A standalone WebAssembly development environment for the internet of things. In *Proceedings of the 21st International Conference on Web Engineering*, pages 353–360, Biarritz, France. Springer.

[Le Pochat et al. 2019] Le Pochat, V., Van Goethem, T., Tajalizadehkhoob, S., Korczyński, M., and Joosen, W. (2019). Tranco: A research-oriented top sites ranking hardened against manipulation. In *26th Annual Network and Distributed System Security Symposium*, NDSS '19. `10.14722/ndss.2019.23386`.

[Lehmann et al. 2020] Lehmann, D., Kinder, J., and Pradel, M. (2020). Everything old is new again: Binary security of WebAssembly. In *Proceedings of the 29th USENIX Security Symposium*, pages 217–234, Virtual Event. USENIX Association.

[Lehmann and Pradel 2019] Lehmann, D. and Pradel, M. (2019). Wasabi: A framework for dynamically analyzing WebAssembly. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1045–1058, Providence, RI, USA. ACM.

[Lehmann and Pradel 2022] Lehmann, D. and Pradel, M. (2022). Finding the dwarf: Recovering precise types from WebAssembly binaries. In *Proceedings of the 43rd International Conference on Programming Language Design and Implementation*, pages 410–425, San Diego, CA, USA. ACM.

[Lehmann et al. 2021] Lehmann, D., Torp, M. T., and Pradel, M. (2021). Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of WebAssembly.

[Levick 2024] Levick, R. (2024). Deconstructing webassembly components. `https://www.youtube.com/@wasmio`.

[Li et al. 2021] Li, B., Fan, H., Gao, Y., and Dong, W. (2021). ThingSpire OS: A WebAssembly-based IoT operating system for cloud-edge integration. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pages 487–488, Virtual Event. ACM.

[Li and Zhang 2022] Li, L. T. and Zhang, M. (2022). Poster: EOSDFA: Data flow analysis of EOSIO smart contracts. In *Proceedings of the Conference on Computer and Communications Security*, pages 3391–3393, Los Angeles, CA, USA. ACM.

[Liang et al. 2018] Liang, H., Pei, X., Jia, X., Shen, W., and Zhang, J. (2018). Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218.

[Liu et al. 2021] Liu, R., Garcia, L., and Srivastava, M. (2021). Aerogel: Lightweight access control framework for WebAssembly-based bare-metal IoT devices. In *Proceedings of the 6th Symposium on Edge Computing*, pages 94–105, San Jose, CA, USA. IEEE.

[LLVM 2023] LLVM (2023). The LLVM compiler infrastructure. `https://github.com/llvm/llvm-project`.

[LLVM-Team 2023] LLVM-Team (2023). The LLVM project. `https://llvm.org/`.

[Marques et al. 2022] Marques, F., Fragoso Santos, J., Santos, N., and Adão, P. (2022). Concolic execution for WebAssembly. In *Proceedings of the 36th European Conference on Object-Oriented Programming*, Berlin, Germany. DROPS.

[Mazaheri et al. 2022] Mazaheri, M. E., Bayat Sarmadi, S., and Taheri Ardakani, F. (2022). A study of timing side-channel attacks and countermeasures on JavaScript and WebAssembly. *The ISC International Journal of Information Security*, 14(1):27–46.

[McFadden et al. 2018] McFadden, B., Lukasiewicz, T., Dileo, J., and Engler, J. (2018). Security chasms of WASM. *NCC Group Whitepaper*.

[Ménétrey et al. 2021] Ménétrey, J., Pasin, M., Felber, P., and Schiavoni, V. (2021). Twine: An embedded trusted runtime for WebAssembly. In *Proceedings of the 37th International Conference on Data Engineering*, pages 205–216, Chania, Greece. IEEE.

[Ménétrey et al. 2022] Ménétrey, J., Pasin, M., Felber, P., and Schiavoni, V. (2022). WebAssembly as a common layer for the cloud-edge continuum. In *Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge*, pages 3–8, Minneapolis, MN, USA. ACM.

[Michael et al. 2023] Michael, A. E., Gollamudi, A., Bosamiya, J., Disselkoen, C., Denlinger, A., Watt, C., Parno, B., Patrignani, M., Vassena, M., and Stefan, D. (2023). MSWasm: Soundly enforcing memory-safe execution of unsafe code. *Proceedings of the ACM on Programming Languages*, 1(1).

[Microsoft 1996] Microsoft (1996). Microsoft announces activex technologies. `https://news.microsoft.com/1996/03/12/microsoft-announces-activex-technologies/`.

[Ming et al. 2016] Ming, J., Wu, D., Wang, J., Xiao, G., and Liu, P. (2016). StraightTaint: Decoupled offline symbolic taint analysis. In *Proceedings of the 31st International Conference on Automated Software Engineering*, pages 308–319, Singapore. IEEE.

[Munsters et al. 2021] Munsters, A., Pupo, A. L. S., Bauwens, J., and Boix, E. G. (2021). Oron: Towards a dynamic analysis instrumentation platform for AssemblyScript. In *Proceedings of the 5th International Conference on the Art, Science, and Engineering of Programming*, pages 6–13, Cambridge, UK. ACM.

[Musch et al. 2019] Musch, M., Wressnegger, C., Johns, M., and Rieck, K. (2019). New kid on the web: A study on the prevalence of webassembly in the wild. In *Proceedings of the 16th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–42, Gothenburg, Sweden. Springer.

[Ménétrey et al. 2022] Ménétrey, J., Pasin, M., Felber, P., and Schiavoni, V. (2022). WaTZ: A trusted WebAssembly runtime environment with remote attestation for TrustZone. In *Proceedings of the 42nd International Conference on Distributed Computing Systems*, pages 1177–1189, Bologna, Italy. IEEE.

[Narayan et al. 2020] Narayan, S., Disselkoen, C., Garfinkel, T., Froyd, N., Rahm, E., Lerner, S., Shacham, H., and Stefan, D. (2020). Retrofitting fine grain isolation in the Firefox renderer. In *Proceedings of the 29th USENIX Security Symposium*, pages 699–716, Boston, MA, USA. USENIX Association.

[Narayan et al. 2021] Narayan, S., Disselkoen, C., Moghimi, D., Cauligi, S., Johnson, E., Gang, Z., Vahldiek-Oberwagner, A., Sahita, R., Shacham, H., Tullsen, D., et al. (2021). Swivel: Hardening WebAssembly against Spectre. In *Proceedings of the 30th USENIX Security Symposium*, pages 1433–1450, Vancouver, BC, Canada. USENIX Association.

[Narayan et al. 2019] Narayan, S., Garfinkel, T., Lerner, S., Shacham, H., and Stefan, D. (2019). Gobi: WebAssembly as a practical path to library sandboxing. arXiv preprint 1912.02285. 10.48550/arXiv.1912.02285.

[Nieke et al. 2021] Nieke, M., Almstedt, L., and Kapitza, R. (2021). Edgedancer: Secure mobile WebAssembly services on the edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, pages 13–18, Virtual Event. ACM.

[Petrov et al. 2020] Petrov, I., Invernizzi, L., and Bursztein, E. (2020). CoinPolice: Detecting hidden cryptojacking attacks with neural networks. arXiv preprint 2006.10861. 10.48550/arXiv.2006.10861.

[Pop et al. 2022] Pop, V. A. B., Niemi, A., Manea, V., Rusanen, A., and Ekberg, J.-E. (2022). Towards securely migrating WebAssembly enclaves. In *Proceedings of the 15th European Workshop on Systems Security*, pages 43–49, Rennes, France. ACM.

[Protzenko et al. 2019] Protzenko, J., Beurdouche, B., Merigoux, D., and Bhargavan, K. (2019). Formally verified cryptographic Web applications in WebAssembly. In *Proceedings of the 40th Symposium on Security and Privacy*, pages 1256–1274, San Francisco, CA, USA. IEEE.

[Qiang et al. 2018] Qiang, W., Dong, Z., and Jin, H. (2018). Se-Lambda: Securing privacy-sensitive serverless applications using SGX enclave. In *Proceedings of the 14th International Conference on Security and Privacy in Communication Systems*, pages 451–470, Singapore. Springer.

[Quan et al. 2019] Quan, L., Wu, L., and Wang, H. (2019). EVulHunter: Detecting fake transfer vulnerabilities for EOSIO's smart contracts at Webassembly-level. 10.48550/arXiv.1906.10362.

[Radovici et al. 2018] Radovici, A., Rusu, C., and Serban, R. (2018). A survey of IoT security threats and solutions. In *Proceedings of the 17th RoEduNet Conference: Networking in Education and Research*, pages 1–5, Cluj-Napoca, Romania. IEEE.

[Renner et al. 2018] Renner, J., Cauligi, S., and Stefan, D. (2018). Constant-time WebAssembly. In *Proceedings of the 45th Symposium on Principles of Programming Languages*, Los Angeles, CA, USA. ACM.

[Rokicki 2022] Rokicki, T. (2022). *Side Channels in Web Browsers: Applications to Security and Privacy*. Thèse de doctorat, Institut National des Sciences Appliquées de Rennes. https://hal.science/tel-04493651/.

[Romano et al. 2022] Romano, A., Lehmann, D., Pradel, M., and Wang, W. (2022). Wobfuscator: Obfuscating JavaScript malware via opportunistic translation to WebAssembly. In *Proceedings of the 43rd Symposium on Security and Privacy*, pages 1574–1589, San Francisco, CA, USA. IEEE.

[Romano et al. 2021] Romano, A., Liu, X., Kwon, Y., and Wang, W. (2021). An empirical study of bugs in WebAssembly compilers. In *Proceedings of the 36th International Conference on Automated Software Engineering*, pages 42–54, Melbourne, Australia. IEEE.

[Romano and Wang 2020a] Romano, A. and Wang, W. (2020a). Wasim: Understanding WebAssembly applications through classification. In *Proceedings of the 35th International Conference on Automated Software Engineering*, pages 1321–1325, Melbourne, Australia. IEEE.

[Romano and Wang 2020b] Romano, A. and Wang, W. (2020b). WasmView: Visual testing for WebAssembly applications. In *Proceedings of the 42nd International Conference on Software Engineering*, pages 13–16, Seoul, South Korea. ACM.

[Rossberg 2024] Rossberg, A. (2024). WebAssembly specification. `https://webassembly.github.io/spec/core/index.html`.

[Rourke 2018] Rourke, M. (2018). *Learn WebAssembly: Build web applications with native performance using Wasm and C/C++*. Packt Publishing Ltd.

[Ruth et al. 2022] Ruth, K., Kumar, D., Wang, B., Valenta, L., and Durumeric, Z. (2022). Toppling top lists: Evaluating the accuracy of popular website lists. In *Proceedings of the 22nd Internet Measurement Conference*, pages 374–387, Nice, France. ACM.

[Shirey 2007] Shirey, R. (2007). Internet security glossary, version 2. RFC 4949. `https://www.rfc-editor.org/info/rfc4949`.

[Sletten 2022] Sletten, B. (2022). *WebAssembly: The Definitive Guide*. O'Reilly Media, Inc.

[Spreitzer et al. 2017] Spreitzer, R., Moonsamy, V., Korak, T., and Mangard, S. (2017). Systematic classification of side-channel attacks: A case study for mobile devices. *IEEE Communications Surveys & Tutorials*, 20(1):465–488.

[stackoverflow 2023] stackoverflow (2023). Stackoverflow. `https://stackoverflow.com/`.

[Stephen 2022] Stephen, A. (2022). Awesome WebAssembly languages. `https://github.com/appcypher/awesome-wasm-langs`.

[Stiévenart et al. 2022a] Stiévenart, Q., Binkley, D. W., and De Roover, C. (2022a). Static stack-preserving intra-procedural slicing of WebAssembly binaries. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2031–2042, Pittsburgh, PA, USA. IEEE.

[Stiévenart and De Roover 2020] Stiévenart, Q. and De Roover, C. (2020). Compositional information flow analysis for WebAssembly programs. In *Proceedings of the 20th International Working Conference on Source Code Analysis and Manipulation*, pages 13–24, Adelaide, Australia. IEEE.

[Stiévenart et al. 2021] Stiévenart, Q., De Roover, C., and Ghafari, M. (2021). The security risk of lacking compiler protection in WebAssembly. In *Proceedings of the 21st International Conference on Software Quality, Reliability and Security*, pages 132–139, Hainan, China. IEEE.

[Stiévenart et al. 2022b] Stiévenart, Q., De Roover, C., and Ghafari, M. (2022b). Security risks of porting C programs to WebAssembly. In *Proceedings of the 37th Symposium on Applied Computing*, pages 1713–1722, Virtual Event. ACM.

[Stock et al. 2017] Stock, B., Johns, M., Steffens, M., and Backes, M. (2017). How the Web tangled itself: Uncovering the history of client-side Web (in)security. In *Proceedings of the 26th USENIX Security Symposium*, pages 971–987, Vancouver, BC, Canada. USENIX Association.

[Sun et al. 2019] Sun, J., Cao, D., Liu, X., Zhao, Z., Wang, W., Gong, X., and Zhang, J. (2019). SELWasm: A code protection mechanism for WebAssembly. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing with Applications*, pages 1099–1106, Xiamen, China. IEEE.

[Sun et al. 2022] Sun, S., Ma, H., Song, Z., and Zhang, R. (2022). WebCloud: Web-based cloud storage for secure data sharing across platforms. *IEEE Transactions on Dependable and Secure Computing*, 19(3):1871–1884.

[Szanto et al. 2018] Szanto, A., Tamm, T., and Pagnoni, A. (2018). Taint tracking for WebAssembly. *arXiv preprint arXiv:1807.08349*.

[Titzer 2022] Titzer, B. L. (2022). A fast in-place interpreter for WebAssembly. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2).

[Tiwari et al. 2018] Tiwari, T., Starobinski, D., and Trachtenberg, A. (2018). Distributed Web mining of Ethereum. In *Proceedings of the International Symposium on Cyber Security Cryptography and Machine Learning*, pages 38–54, Beer-Sheva, Israel. Springer.

[Tsoupidi et al. 2021] Tsoupidi, R. M., Balliu, M., and Baudry, B. (2021). Vivienne: Relational verification of cryptographic implementations in WebAssembly. In *Proceedings of the Secure Development Conference*, pages 94–102, Atlanta, GA, USA. IEEE.

[Vassena et al. 2021] Vassena, M., Disselkoen, C., Gleissenthall, K. v., Cauligi, S., Kıcı, R. G., Jhala, R., Tullsen, D., and Stefan, D. (2021). Automatically eliminating speculative leaks from cryptographic code with Blade. *Proceedings of the ACM on Programming Languages*, 5(POPL).

[Vassena and Patrignani 2020] Vassena, M. and Patrignani, M. (2020). Memory safety preservation for WebAssembly. In *Proceedings of the 47th Symposium on Principles of Programming Languages*, New Orleans, LA, USA. ACM.

[W3C Community Group 2022a] W3C Community Group (2022a). WA: Security. `https://webassembly.org/docs/security/`.

[W3C Community Group 2022b] W3C Community Group (2022b). WebAssembly. https://webassembly.org.

[W3Techs 2023] W3Techs (2023). Usage statistics of JavaScript as client-side programming language on websites. https://w3techs.com/technologies/details/cp-javascript.

[Wang et al. 2019] Wang, S., Ye, G., Li, M., Yuan, L., Tang, Z., Wang, H., Wang, W., Wang, F., Ren, J., Fang, D., and Wang, Z. (2019). Leveraging WebAssembly for numerical JavaScript code virtualization. *IEEE Access*, 7:182711–182724.

[Wang 2021] Wang, W. (2021). Empowering web applications with WebAssembly: are we there yet? In *Proceedings of the 36th International Conference on Automated Software Engineering*, pages 1301–1305, Melbourne, Australia. IEEE.

[Wang et al. 2018] Wang, W., Ferrell, B., Xu, X., Hamlen, K. W., and Hao, S. (2018). SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks. In *Proceedings of the 23rd European Symposium on Research in Computer Security*, pages 122–142, Barcelona, Spain. Springer.

[Watt et al. 2019a] Watt, C., Renner, J., Popescu, N., Cauligi, S., and Stefan, D. (2019a). CT-Wasm: Type-driven secure cryptography for the Web ecosystem. *Proceedings of the ACM on Programming Languages*, 3(POPL).

[Watt et al. 2019b] Watt, C., Rossberg, A., and Pichon-Pharabod, J. (2019b). Weakening WebAssembly. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–28.

[WebAssembly 2024] WebAssembly (2024). Webassembly proposals. https://github.com/WebAssembly/proposals.

[Wen and Weber 2020] Wen, E. and Weber, G. (2020). Wasmachine: Bring the edge up to speed with a WebAssembly OS. In *Proceedings of the 13th International Conference on Cloud Computing*, pages 353–360, Beijing, China. IEEE.

[Wen et al. 2021] Wen, E., Weber, G., and Nanayakkara, S. (2021). WasmAndroid: A cross-platform runtime for native programming languages on Android (WIP paper). In *Proceedings of the 22nd International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 80–84, Virtual Event. ACM.

[Will et al. 2021] Will, N. C., Heinrich, T., Viescinski, A. B., and Maziero, C. A. (2021). Trusted inter-process communication using hardware enclaves. In *Proceedings of the 15th International Systems Conference*, pages 1–7, Vancouver, BC, Canada. IEEE.

[Xie et al. 2022] Xie, Q., Tang, S., Zheng, X., Lin, Q., Liu, B., Duan, H., and Li, F. (2022). Building an open, robust, and stable voting-based domain top list. In *Proceedings of the 31st USENIX Security Symposium*, pages 625–642, Boston, MA, USA. USENIX Association.

[Yan et al. 2021] Yan, Y., Tu, T., Zhao, L., Zhou, Y., and Wang, W. (2021). Understanding the performance of WebAssembly applications. In *Proceedings of the 21st Internet Measurement Conference*, pages 533–549, Virtual Event. ACM.

[Yee et al. 2010] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. (2010). Native Client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53(1):91–99.

[Yu et al. 2020] Yu, G., Yang, G., Li, T., Han, X., Guan, S., Zhang, J., and Gu, G. (2020). MinerGate: A novel generic and accurate defense solution against Web based cryptocurrency mining attacks. In *Proceedings of the 17th China Cyber Security Annual Conference*, pages 50–70, Beijing, China. Springer.

[Zakai 2011] Zakai, A. (2011). Emscripten: An LLVM-to-JavaScript compiler. In *Proceedings of the International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, pages 301–312, Portland, Oregon, USA. ACM.

[Zhang et al. 2024] Zhang, Y., Zheng, S., Wang, H., Wu, L., Huang, G., and Liu, X. (2024). VM matters: A comparison of WASM VMs and EVMs in the performance of blockchain smart contracts. *ACM Transactions on Modeling and Performance Evaluation of Computer Systems*, 9(2).