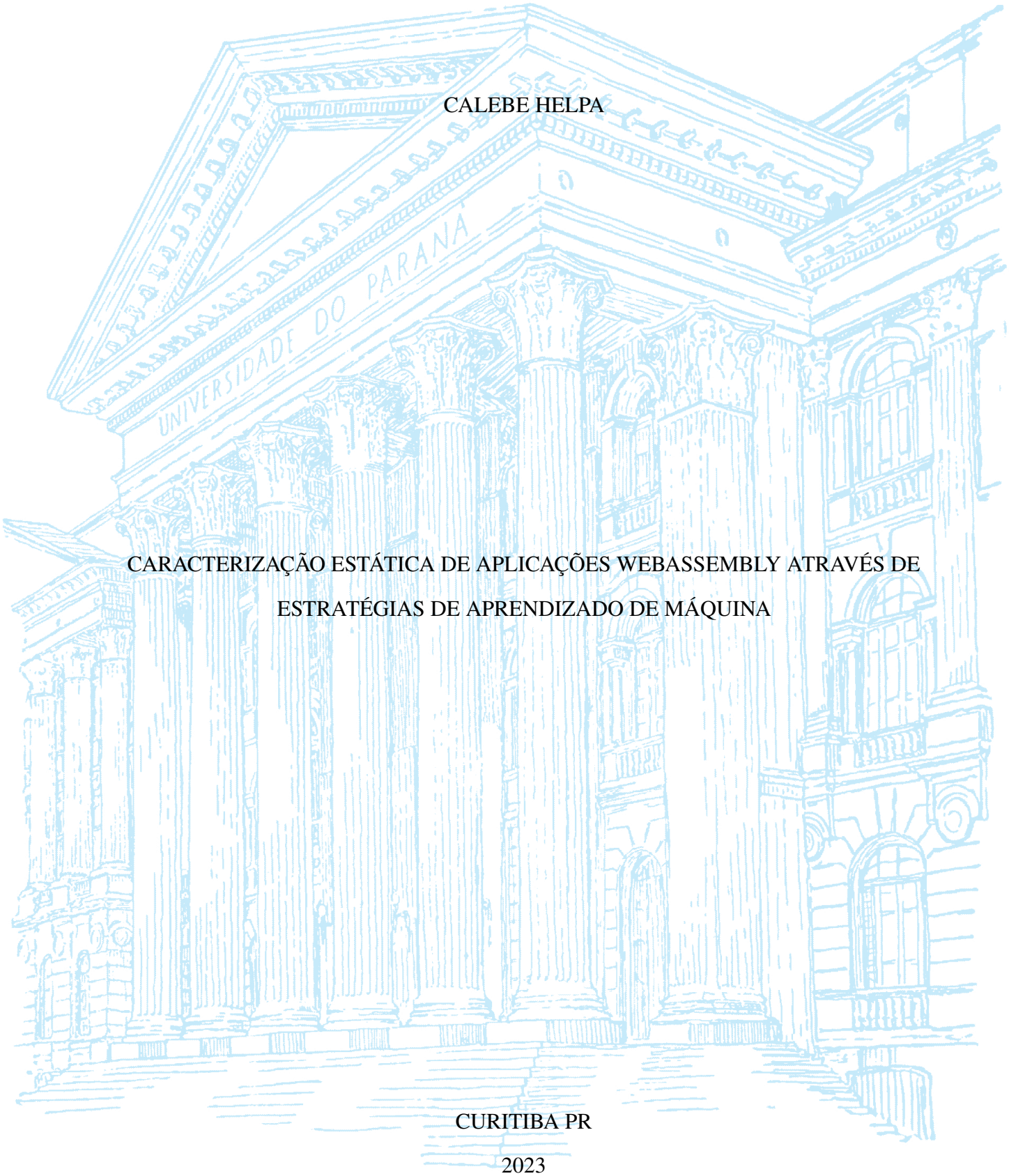
UNIVERSIDADE FEDERAL DO PARANÁ

CALEBE HELPA



CARACTERIZAÇÃO ESTÁTICA DE APLICAÇÕES WEBASSEMBLY ATRAVÉS DE ESTRATÉGIAS DE APRENDIZADO DE MÁQUINA

CURITIBA PR

2023

CALEBE HELPA

CARACTERIZAÇÃO ESTÁTICA DE APLICAÇÕES WEBASSEMBLY ATRAVÉS DE ESTRATÉGIAS DE APRENDIZADO DE MÁQUINA

Trabalho apresentado como requisito à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Carlos Alberto Maziero.

Coorientador: Tiago Heinrich.

CURITIBA PR

2023

Ficha catalográfica

Substituir o arquivo `0-iniciais/catalografica.pdf` pela ficha catalográfica fornecida pela Biblioteca da UFPR (PDF em formato A4).

Instruções para obter a ficha catalográfica e fazer o depósito legal da tese/dissertação (contribuição de André Hochuli, abril 2019):

1. Verificar se está usando a versão mais recente do modelo do PPGInf e atualizar, se for necessário (`https://gitlab.c3sl.ufpr.br/maziero/tese`).
2. conferir o Checklist de formato do Sistema de Bibliotecas da UFPR, em `https://portal.ufpr.br/teses_servicos.html`.
3. Enviar email para "`referencia.bct@gmail.com`" com o arquivo PDF da dissertação/tese, solicitando a respectiva ficha catalográfica.
4. Ao receber a ficha, inseri-la em seu documento (substituir o arquivo `0-iniciais/catalografica.pdf` do diretório do modelo).
5. Emitir a Certidão Negativa (CND) de débito junto a biblioteca (`https://www.portal.ufpr.br/cnd.html`).
6. Avisar a secretaria do PPGInf que você está pronto para o depósito. Eles irão mudar sua titulação no SIGA, o que irá liberar uma opção no SIGA pra você fazer o depósito legal.
7. Acesse o SIGA (`http://www.prppg.ufpr.br/siga`) e preencha com cuidado os dados solicitados para o depósito da tese.
8. Aguarde a confirmação da Biblioteca.
9. Após a aprovação do pedido, informe a secretaria do PPGInf que a dissertação/tese foi depositada pela biblioteca. Será então liberado no SIGA um link para a confirmação dos dados para a emissão do diploma.

Ficha de aprovação

Substituir o arquivo 0-iniciais/aprovacao.pdf pela ficha de aprovação fornecida pela secretaria do programa, em formato PDF A4.

AGRADECIMENTOS

Gostaria de expressar minha gratidão a todas as pessoas que contribuíram de maneira significativa para a realização deste trabalho de conclusão de curso. Primeiramente, quero agradecer aos meus pais, Gilson e Juliana, por seu amor, apoio constante e incentivo ao longo de minha jornada acadêmica. Seu comprometimento e confiança em mim foram fundamentais para que eu chegasse até aqui.

À minha querida noiva, Jéssica, quero dedicar um sincero agradecimento. Seu apoio emocional, paciência e compreensão foram essenciais durante os momentos desafiadores deste percurso. Sua presença constante me inspirou a buscar a excelência e a superar obstáculos, e por isso, agradeço de todo o coração.

Ao meu orientador, Dr. Carlos Alberto Maziero, expresso minha profunda admiração e gratidão. Sua orientação, conhecimento acadêmico e dedicação foram vitais para o meu desenvolvimento ao longo de todo o curso. Seus ensinos moldaram minhas ideias e direcionaram meu caminho, e por isso, sou muito grato.

Por fim, não posso deixar de mencionar meu coorientador, Tiago Heinrich, cuja instrução e colaboração enriqueceram sobremaneira este projeto. Sua experiência e disposição em compartilhar conhecimentos foram um presente valioso, e sou grato pela sua participação ativa em todas as etapas deste trabalho.

RESUMO

Nos últimos anos, o aumento no uso de aplicações web tem impulsionado o desenvolvimento de novas tecnologias, incluindo o WebAssembly. Com a necessidade crescente de processamento e segurança ao navegar pela internet, o formato de instruções binárias WebAssembly tem ganhado notoriedade, sendo adotado atualmente em mais de 95% dos navegadores. No entanto, o aumento no uso desse formato também trouxe preocupações em relação à sua segurança e sua possível utilização de forma maliciosa. Dado que o WebAssemby é um formato de instruções de baixo nível, torna-se essencial a identificação do propósito dos códigos desenvolvidos, por meio da extração de suas características. Nesse contexto, esse trabalho apresenta uma estratégia para a classificação de binários WebAssembly, utilizando a extração de características encontradas. Para a classificação e posterior identificação de possíveis ameaças, serão utilizados métodos de aprendizado de máquina treinados com as características extraídas dos binários. Por meio dessa abordagem será possível classificar os binários de maneira automatizada, e identificar possíveis ameaças, trazendo maior segurança para a utilização de módulos WebAssembly.

Palavras-chave: Caracterização, Classificação de binários, e WebAssembly.

ABSTRACT

In recent years, the increase in the use of web applications has driven the development of new technologies, including WebAssembly. With the growing need for processing power and security when browsing the internet, the WebAssembly binary instruction format has gained notoriety, currently being adopted in more than 95% of browsers. However, the increased use of this format has also brought concerns regarding its security and possible malicious use. Given that WebAssembly is a low-level instruction format, identifying the purpose of the developed code becomes essential through the extraction of its characteristics. In this context, this work presents a strategy for classifying WebAssembly binaries, using the extraction of characteristics found in them. To classify and identify possible threats, machine learning methods will be used, trained with the characteristics extracted from the binaries. Through this approach, it will be possible to classify the binaries in an automated way and identify possible threats, bringing greater security to the use of WebAssembly modules.

Keywords: Characterization, Binary Classification, and WebAssembly.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

CPG Code Property Graph

CPU Central Processing Unit

IP Internet Protocol

JS JavaScript

ML Machine Learning

SUMÁRIO

1 INTRODUÇÃO

Com o constante desenvolvimento de aplicações web e a crescente necessidade de processamentos mais rápidos, o formato de instruções WebAssembly tem ganhado notoriedade (Romano et al., 2022). Atualmente a principal linguagem utilizada para desenvolver aplicações que rodam em páginas web é o JavaScript (JS), porém devido o aumento da quantidade e complexidade dos programas nos navegadores os algoritmos precisam de maior desempenho para serem executados, exigindo mais dos processadores e tornando maior a espera pelo carregamento da página. Inserido nesse contexto então surge o WebAssembly, um formato de instruções em binário que roda em máquinas virtuais.

O WebAssembly tem como objetivo dar suporte para as aplicações Web, tornando mais rápido o processamento de componentes que demandam mais da Central Processing Unit (CPU) e diminuindo o uso de memória para carregar a página (Falliere, 2018). Concomitantemente, o WebAssembly possui garantia de funcionamento em diferentes navegadores (Romano et al., 2022). Por utilizar um ambiente virtualizado (*sandbox*) para serem executadas, as aplicações em WebAssembly possuem alto grau de portabilidade, facilitando dessa forma a reutilização de códigos e trazendo segurança de funcionamento em diferentes plataformas.

Contudo, apesar de seus benefícios, o WebAssembly também possui suas vulnerabilidades. Por ser uma tecnologia recente ainda existem muitas vulnerabilidades de segurança e espaços para melhorias a serem explorados. Como explorado em (Michael et al., 2022), problemas como o de *buffer overflow* são presentes em casos de interação entre JS e WebAssembly. Por conta da utilização de um *buffer* gerenciado implementado por meio de um vetor para ser utilizado como memória do programa, problemas relacionados à armazenamento de dados do programa também estão presentes, como explorado em (Lehmann et al., 2020). A fim de mitigar tais riscos, estudos recentes focam em corrigir falhas de design, adicionar novos recursos e melhorar recursos já implantados. Avaliando os problemas relacionados à memória, (Michael et al., 2022) apresenta duas implementações de extensão para segurança de memória. Outra proposta de melhoria é feita por (Bosamiya et al., 2022), que desenvolve dois compiladores utilizando diferentes técnicas, a fim de melhorar a segurança relacionada à interação entre os binários WebAsesembly e os *hosts* em que estão rodando, de modo que os módulos gerados não são capazes de acessar sessões de memória do *host* ou desviar o fluxo de execução para locais não permitidos.

Outro problema relacionado a utilização má intencionada do WebAssembly é a implementação de *malwares* que realizam *cryptojacking* por meio de navegadores (Naseem et al., 2021). Módulos WebAssembly com esse tipo de *malware* executam mineração de criptomoeda de forma secreta e não autorizada pelos usuários. Diferentes aplicações foram desenvolvidas a fim de detectar a presença desse tipo de ataque, um exemplo é o MINOS, descrito em (Naseem et al., 2021). De modo geral essa ferramenta converte o arquivo binário em uma imagem e realiza a detecção da presença de um ataque de *criptojacking* por meio de redes neurais convolucionais (Naseem et al., 2021). Contudo, esta é apenas uma das ameaças no momento que atacantes exploram WebAssembly para efetuar ataques.

Visando a segurança no ambiente Web e a segurança de aplicações WebAssembly. A proposta deste trabalho é o desenvolvimento de uma estratégia estática para a caracterização de aplicações WebAssembly, através do uso de informações encontradas no binário WebAssembly. Esta estratégia utiliza as informações encontradas nos binários para a posterior classificação das aplicações e detecção de intrusão de aplicações WebAssembly.

1.1 OBJETIVOS

Objetivo geral: Realizar um estudo voltado à caracterização de binários WebAssembly. Primeiramente averiguando o uso das informações encontradas no binários para a classificação das aplicações, que posteriormente serão utilizadas para a detecção de intrusão.

Objetivos específicos:

- Explorar ferramentas para a extração de informações em binários WebAssembly;
- Definir características que sejam representativas dos binários e possibilitem a classificação de binários WebAssembly;
- Realizar a análise através de modelos de aprendizagem de máquina em um conjunto de dados de binários WebAssembly com base nas características encontradas; e
- Definir o comportamento dos binários e analisar o desempenho da análise com base nos resultados obtidos.

1.2 METODOLOGIA

Este trabalho de conclusão de curso apresenta um estudo sobre a utilização de binários WebAssembly para realizar a classificação e detecção de intrusão. A pesquisa envolveu a investigação de ferramentas para extração de características de binários WebAssembly, bem como uma revisão do estado da arte para identificar estratégias estáticas existentes para a classificação e detecção de intrusão com base em binários WebAssembly. Em seguida, foram explorados diferentes conjuntos de dados para criar uma base de dados e extrair características dos binários, além de treinar modelos de aprendizagem de máquina para a classificação das aplicações WebAssembly. Após essa etapa, foi possível avaliar o processo de classificação de binários WebAssembly utilizando as informações extraídas. O trabalho concluiu com a validação da abordagem proposta, demonstrando a eficácia do uso de técnicas de ML na detecção de intrusões em binários WebAssembly.

1.3 ORGANIZAÇÃO TEXTUAL

Este trabalho está dividido em 6 capítulos. O capítulo 2 descreve a fundamentação teórica, apresentando os conceitos fundamentais de WebAssembly, sistema de detecção de intrusão, detecção de anomalia e ML. O capítulo 3 apresenta a revisão bibliográfica da literatura envolvendo detecção de anomalias e medidas de segurança para WebAssembly. O capítulo 4 apresenta a proposta, descrevendo a estratégia, a coleta de dados e o método de avaliação. O capítulo 5 apresenta a avaliação dos resultados, especificando o conjunto de testes, a extração de dados, o dataset proposto e os resultados obtidos. Por fim, o capítulo 6 finaliza com os principais avanços alcançados e apresenta sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo introduz os conceitos necessários para a compreensão e desenvolvimento da proposta deste trabalho. A Seção 2.1 apresenta os conceitos básicos para o entendimento da linguagem WebAssembly, a Seção 2.2 apresenta conceitos relacionados à detecção de intrusão e a seção 2.3 descreve detecção de anomalias.

2.1 WEBASSEMBLY

WebAssembly é uma linguagem de instruções binárias que vêm conquistando espaço no ramo de desenvolvimento web (Lehmann et al., 2020); atualmente, em Março de 2023, aproximadamente 95% dos navegadores possuem suporte para a mesma. O formato de instruções é executado em máquinas virtuais, de forma a trazer portabilidade para que o mesmo código possa ser utilizado em diferentes arquiteturas de sistemas (Battaglineg, 2021). Algoritmos em diferentes linguagens como C, C++ e Rust podem ser compilados de tal maneira e utilizados como módulos em navegadores de internet para executar tarefas com alta performance. Por esse motivo, empresas como Google, eBay e Norton têm adotado o WebAssembly para realizar tarefas custosas com rapidez (Romano e Wang, 2020). Para desenvolvedores a flexibilidade oferecida para portar códigos de linguagens que não sejam desenvolvidas para a Web, flexibiliza o desenvolvimento de novas aplicações no ambiente Web.

Módulos no formato WebAssembly contêm definições de funções, variáveis globais, memórias lineares e tabelas de chamadas indiretas. Funções e variáveis, bem como outros elementos do programa, são identificados por índices representados por números inteiros (Lehmann et al., 2020).

As máquinas virtuais que rodam os módulos WebAssembly seguem o modelo *Último a entrar - Primeiro a sair*, conhecido como pilha. As instruções retiram da pilha de valores suas entradas e inserem na pilha suas saídas. Desta maneira, módulos WebAssembly não utilizam registradores, apenas a pilha de avaliação, variáveis globais, que podem ser utilizadas por todas as funções no módulo, e variáveis locais, que são apenas utilizadas pela função que as declarou (Lehmann et al., 2020).

Sobre os tipos de dados utilizados, apenas quatro tipos primitivos são suportados. Podem ser: *i32*, *i64*, *f32*, *f64*, representando um inteiro de 32 ou 64 bits ou um número em ponto flutuante de 32 ou 64 bits, respectivamente. Esses dados podem ser armazenados em uma memória linear implementada por meio de um vetor de bytes, que por sua vez, só pode ser acessada de forma indireta por meio de índices.

A linguagem WebAssembly como formato de instruções em código binário pode ser depurada por meio de conversores que tornam o código de máquina legível, porém devido ao nível de complexidade a análise manual de tais módulos é impraticável sem ferramentas para fazer esse exame. Essas ferramentas estão disponíveis e tornam mais praticável a análise das seções do código WebAssembly (Falliere, 2018). Um módulo WebAssembly comumente possui três seções: Preâmbulo, Padrão e Customizada (Kim et al., 2022). A seguir são apresentadas as seções que compõem um módulo WebAssembly e suas respectivas especificidades.

2.1.1 Preâmbulo

A seção de preâmbulo possui a indicação do ponto de início de um módulo binário de WebAssembly (Kim et al., 2022). O preâmbulo contém 4 bytes com número mágico do WebAssembly (“`\0asm`”), e um campo de versão (Rossberg, 2018)).

2.1.2 Padrão

Após o preâmbulo temos a seção padrão. Todo módulo possui ao menos uma seção padrão que é validada antes da execução do módulo. Nela estão todas as informações de códigos, funções e memória necessárias para o funcionamento correto das instruções. Na seção padrão temos diferentes subseções, abaixo iremos explorá-las na ordem que aparecem (Kim et al., 2022).

Tipo: A subseção Tipo define o conjunto de tipos de dados que podem ser usados em módulos WebAssembly. Ela inclui tipos numéricos inteiros e de ponto flutuante, bem como tipos de referência, como funções e estruturas. A seção Tipo é essencial para a execução de módulos WebAssembly, pois define como os valores são passados entre as funções e como as funções são chamadas por meio de suas respectivas assinaturas. Em uma assinatura de função estão presentes os tipos dos dados de entrada e o tipo do dado de saída. Estes valores podem ser nulos caso a função não possua entrada ou saída e diferentes funções podem compartilhar a mesma assinatura (Falliere, 2018). Segue um exemplo de assinatura que poderia ser encontrada na subseção Tipo de um módulo WebAssembly:
`(func (param i32 i32) (result i32))`
Neste exemplo, está sendo definida uma função que recebe dois parâmetros inteiros de 32 bits e retorna um valor inteiro de 32 bits. O parâmetro “param” é utilizado para especificar os tipos e a ordem respectiva dos parâmetros da função, o parâmetro “result” é utilizado para especificar o tipo de valor de retorno da função.

Importação: A subseção Importação do WebAssembly é responsável por definir como os módulos WebAssembly podem importar funções e valores externos de outras fontes, como o host do ambiente de execução, para que funcione corretamente (Falliere, 2018). Os dados importados podem variar entre funções, tabelas, memória e variáveis globais que são importados para compartilhar informações com outros módulos e funcionar corretamente (Kim et al., 2022). É por meio dela que módulos WebAssembly podem ser integrados a outros sistemas ou bibliotecas, usando funções e valores definidos fora do próprio módulo (Falliere, 2018). Segue um exemplo simplificado de como a subseção Importação é utilizada em um módulo WebAssembly:
`(import "env" "funcao_externa" (func $funcao_externa (param i32 i32) (result i32)))`
Neste exemplo, está sendo importada uma função chamada “funcao_externa” do módulo externo “env”. O parâmetro “func” é utilizado para indicar que está sendo importada uma função e o nome “$funcao_externa” é utilizado para associar um nome interno à função que está sendo importada.

Função: Uma lista com todas as funções definidas no módulo WebAssembly é estipulada nessa subseção (Kim et al., 2022). Essa lista é armazenada como um vetor de índices que representam os tipos dos campos de dados das funções, ou seja, ela define como as funções são declaradas e definidas em um módulo WebAssembly. As variáveis locais e o corpo das funções estão respectivamente definidos na subseção Código (Rossberg,

2018)). Segue um exemplo simplificado da utilização da subseção Função:

```
(func (export "funcao_exemplo") (param i32 i32) (result
i32)
```

Neste exemplo, é definida a função chamada "funcao_exemplo" que recebe dois parâmetros inteiros de 32 bits e retorna a soma desses parâmetros. O parâmetro "export" é usado para exportar a função para que ela possa ser chamada fora do módulo.

Tabela: Nesta subseção é definido um vetor com valores de referência para acessar as funções de forma indireta. A subseção Tabela define a estrutura da tabela, incluindo seu tipo e tamanho, e fornece instruções para manipular a tabela, como a adição e remoção de elementos e a obtenção do elemento em um índice específico. Desta forma, o WebAssembly consegue garantir maior segurança nos módulos evitando que as funções sejam acessadas de forma direta através de ponteiros (Kim et al., 2022). Segue um exemplo simplificado da utilização da subseção Tabela:

```
(table $tabela 3 funcref)
(elem (i32.const 0) $funcao_1)
```

Neste exemplo, está sendo definida uma tabela chamada "tabela" com um tamanho de 3 elementos, que pode armazenar referências a funções (funcref). por meio do parâmetro "table" é possível declarar a tabela e associar um nome interno a ela. As instruções "elem" são utilizadas para adicionar elementos à tabela. Na primeira linha, a função "funcao_1" está sendo adicionada ao índice 0 da tabela, a instrução "(i32.const 0)" está sendo utilizada para obter o índice e o nome interno "$funcao_1" está sendo utilizado para obter a referência da função.

Memória: Na subseção Memória de um módulo WebAssembly é estabelecido um vetor como Memória Linear de bytes não interpretados. Cada *byte* pode ser lido ou gravado individualmente. A subseção Memória define a estrutura da memória, incluindo seu tipo e tamanho, e fornece instruções para manipular a memória, como a leitura e gravação de bytes em um índice específico. Ela impede o acesso direto à memória, e seu funcionamento é mais um dos elementos essenciais para a garantia de segurança do WebAssembly (Rossberg, 2018). Segue um exemplo simplificado da utilização da subseção Memória:

```
(memory $memoria 1)
(data (i32.const 0) "WebAssembly")
```

Neste exemplo, está sendo definida uma memória chamada "memoria" com um tamanho de 1 página (64KB, tamanho padrão de página de memória em módulos WebAssembly). O parâmetro "memory" é utilizado para declarar a memória e associar um nome interno a ela. A seguir está sendo utilizada a instrução "data" para gravar dados na memória. No exemplo, está sendo gravada a palavra "WebAssembly" no índice 0 da memória, a instrução "(i32.const 0)" está sendo utilizada para obter o índice e a palavra está sendo utilizada como dados.

Código: Como já explorado anteriormente, na subseção Código o corpo das funções e as variáveis são definidas, seguindo a ordem estipulada na declaração das assinaturas das funções na subseção Função. O corpo da função é composto por uma sequência de instruções em formato de *bytecode*, que serão executadas em ordem quando a função for chamada.(Rossberg, 2018). Segue um exemplo simplificado da utilização da subseção Código:

```
(func (export "funcao_exemplo") (param i32 i32) (result
```

```
i32)
(local.get 0)
(local.get 1)
(i32.add)
)
```

Neste exemplo, dentro da função as instruções “(local.get 0)” e “(local.get 1)” são usadas para obter os valores dos parâmetros e colocá-los na pilha, e a instrução “(i32.add)” é usada para realizar a adição na pilha. O resultado da soma é empilhado e retornado usando a instrução “result”, neste exemplo, está sendo retornado um valor do tipo i32.

2.1.3 Customizada

A seção Customizada tem como objetivo fornecer dados necessários para depuração do código. Nela podem estar presentes informações do módulo, símbolos para funções e nomes de variáveis (Kim et al., 2022). Para fins de identificação, a seção Customizada primeiramente possui um nome e em seguida uma sequência de bytes que não são interpretados pela semântica do WebAssembly (Rossberg, 2018), e que, portanto, não introduzem erro (Kim et al., 2022).

2.1.4 Segurança no WebAssembly

O formato de instruções WebAssembly foi projetado com um grande foco em garantir a segurança de seus usuários. Seus principais recursos para certificar sua segurança são: ambiente virtualizado, memória linear e integridade do fluxo de controle (Kim et al., 2022). A seguir iremos explorar com mais detalhes cada um desses recursos.

Ambiente virtualizado: Como abordado anteriormente, os módulos WebAssembly rodam em uma máquina virtual baseada no modelo de pilha. dessa forma o ambiente em que o recurso está rodando não é acessado diretamente. Toda interação de entrada, saída e acesso a recursos do sistema operacional, deve ser realizada através de funções incorporadas pelo WebAssembly, que devem ser importadas pelo módulo. Portanto, o WebAssembly é capaz de estabelecer políticas de segurança aos desenvolvedores e consegue garantir aos usuários que seu ambiente e os recursos do sistema estão sendo acessados pelos módulos de forma limitada e controlada (Rossberg, 2018).

Memória linear: Outra maneira citada anteriormente de certificar sua segurança é por meio do acesso à memória. A memória linear dos módulos WebAssembly são instanciadas em buffers gerenciados, portanto o código pode realizar operações de leitura ou escrita apenas em áreas limitadas da memória (Kim et al., 2022). Outra forma de garantir a segurança através do acesso à memória é por meio da chamada de funções de forma indireta, por meio da tabela citada acima.

Integridade de controle de fluxo: Através de um controle de fluxo estruturado gerado durante o processo de compilação (Kim et al., 2022), módulos WebAssembly são protegidos contra ataques como injeção de *shellcode* ou abuso de saltos irrestritos realizados de forma indireta (Lehmann et al., 2020). Outra proteção que esse recurso traz é a inviabilidade de execução de dados escritos em memória. Todo comportamento que não segue a estrutura definida no processo de compilação resulta em falha, garantindo a segurança do usuário.

O formato de instruções WebAssembly tem se destacado como uma tecnologia promissora para garantir maior segurança em aplicações web. Sua estrutura de ambiente virtualizado, a utilização de memória linear e sua integridade de controle de fluxo trazem benefícios importantes para a proteção contra vulnerabilidades e ataques maliciosos. No entanto, um problema em aberto que merece atenção é a ausência da definição da viabilidade do uso de características para classificação de aplicações WebAssembly. Caso se apresente viável, uma ferramenta capaz de realizar a detecção da utilização do código WebAssembly de maneira maliciosa pode ser o próximo passo na detecção de intrusão em módulos WebAssembly. Para que o WebAssembly se mantenha como uma tecnologia confiável e segura para o desenvolvimento de aplicações web, é necessário que pesquisas e soluções para garantir a segurança durante sua utilização continuem sendo incentivadas.

2.2 SISTEMA DE DETECÇÃO DE INTRUSÃO

A detecção de intrusão é classificada pelo NIST (*National Institute of Standards and Technology*): "O processo de monitorar os eventos que ocorrem em um sistema de computador ou rede e analisá-los para sinais de intrusão, definidos como tentativas de comprometer a confidencialidade, integridade, disponibilidade, ou burlar os mecanismos de segurança de um computador ou rede" (Brown et al., 2002). Atualmente a proeminência de sistemas de detecção de intrusão concentra-se na análise de dados de computadores *host* e de dados que trafegam pela rede, cada uma dessas fontes apresentando suas oportunidades e desafios para manter o ambiente seguro contra intrusões (Brown et al., 2002). A seguir essas duas abordagens são apresentadas em detalhes.

Sistemas de Detecção de Intrusão baseados em Host (HIDS): Esta classe de sistemas de detecção de intrusão tem como alvo monitorar unicamente os dados de uma máquina hospedeira. Para isso, diferentes dados podem ser analisados, como por exemplo chamadas de sistema, registros de *shell*, sistemas de arquivos e eventos de rede (Liu et al., 2018). Cada tipo de dados envolve uma técnica de análise específica, bem como causa diferentes impactos no sistema, implica em volumes de dados distintos a serem processados e compromete conteúdos de dados particulares do sistema (Brown et al., 2002).

Sistemas de Detecção de Intrusão baseados em Rede (NIDS): Sistemas de detecção baseados em Rede operam monitorando os pacotes que trafegam pela rede de determinado sistema. Os principais dados analisados por sistemas dessa classe são pacotes Internet Protocol (IP), cabeçalhos de camada de transporte de segmentos individuais ou até mesmo o conteúdo dos segmentos (Bosamiya et al., 2022). Sistemas de Detecção de Intrusão baseados em Rede operam principalmente em dois tipos de implementação, baseados em assinatura ou baseados em detecção de anomalia (Kumar, 2007).

Neste trabalho iremos adotar o Sistema de Detecção de Intrusão baseado em *Host* a fim de determinar o comportamento dos módulos binários WebAssembly em sua interação com o sistema que está executando o código. Esta estratégia irá utilizar informações extraídas dos binários WebAssembly, para a posterior detecção de anomalias.

2.3 DETECÇÃO DE ANOMALIA

Identificar padrões de dados que não correspondem ao comportamento esperado do sistema é o objetivo de um Sistema de Detecção de Anomalia (Chandola et al., 2009). Para

ser capaz de realizar essa detecção, o sistema deve possuir conhecimento sobre as informações presentes no meio sendo monitorado. Esse meio pode ser a rede de conexão, em que os dados monitorados serão os do tráfego de rede, ou o *host*, nesse caso buscando anomalias em relação ao comportamento considerado normal para o sistema (Liu et al., 2018).

Para Sistemas de Detecção de Intrusão baseados em *host* são as subsequências de instruções que, quando consideradas em conjunto, definem um comportamento anormal do sistema (Chandola et al., 2009). A fim de monitorar o comportamento de códigos ou módulos específicos, duas técnicas podem ser empregadas, análise estática ou dinâmica.

Análise dinâmica é uma técnica de análise de software que permite avaliar o comportamento do programa durante a sua execução. A análise dinâmica é realizada por meio de testes e simulações, o que possibilita identificar erros de programação, comportamentos inesperados e falhas de segurança durante a interação entre o código e o ambiente em que ele é executado (Kirchmayr et al., 2016).

Um exemplo de implementação de análise dinâmica é o uso de ferramentas de depuração de código. Essas ferramentas permitem ao programador acompanhar a execução do programa passo a passo e identificar os trechos de código que estão causando erros ou comportamentos inesperados. Além disso, a análise dinâmica também é utilizada na detecção de vulnerabilidades de segurança em sistemas, por meio de técnicas como *fuzzing* e análise de fluxo de dados (Lehmann e Pradel, 2019).

Já a análise estática é realizada através da extração de características do código, porém sem efetuar sua execução, definindo então uma representação abstrata do comportamento do programa (Kirchmayr et al., 2016). Essa técnica tem sido amplamente utilizada especialmente em sistemas críticos, como os usados em aviação e controle de tráfego aéreo.

A análise estática pode identificar uma série de problemas de segurança, incluindo vulnerabilidades de entrada inválida, de autenticação e gerenciamento de sessão quebrados e problemas de injeção de código malicioso. Além disso, essa técnica pode ser usada para detectar comportamentos anômalos no código, alertando sobre possíveis ataques de *malware* (Pistoia et al., 2007).

Outra aplicação da análise estática é na aplicação na detecção de anomalias em sistemas embarcados, como os usados em carros autônomos e dispositivos médicos (Zhang et al., 2020). A análise estática é capaz de identificar possíveis falhas de segurança em tais sistemas, possibilitando a correção antes que sofram falhas durante a execução em fase de produção (Zhang et al., 2020).

2.4 MACHINE LEARNING

Machine Learning (ML) é um subcampo da Inteligência Artificial dedicado a desenvolver algoritmos e modelos capazes de extrair características e realizar previsões a partir de dados, contudo sem serem explicitamente programados para isso. Com crescente notoriedade, a área de ML tem se mostrado uma ferramenta fundamental para análise, interpretação e tomada de decisões em diferentes campos do conhecimento. No âmbito da segurança, a aplicação de técnicas de ML tem proporcionado avanços significativos, sendo considerada estado da arte para solucionar diferentes problemas em aberto (Ceschin et al., 2020).

De maneira geral, a utilização de técnicas de ML para segurança consiste em uma sequência de etapas bem definidas. Primeiramente deve ser feita a coleta, extração e transformação dos dados para que possam estar em um formato adequado para análise. Em seguida é realizada a fase de treinamento do modelo, na qual os algoritmos são aplicados aos dados preparados anteriormente. Nesta fase, as diferentes técnicas são utilizadas a fim de construir um modelo capaz de aprender características e padrões das ameaças de segurança. Por fim, após o treinamento

ocorre a fase de teste e validação do modelo, na qual dados não utilizados no treinamento são usados para avaliar a capacidade de generalização do modelo e a sua precisão na detecção de ameaças (Ceschin et al., 2020).

Considerando o contexto de aplicação de ML para segurança, diferentes técnicas são empregadas para resolver problemas distintos. Tendo em vista o contexto de detecção de arquivos maliciosos, a técnica de classificação é amplamente utilizada para identificação e categorização de dados em diferentes classes ou rótulos (Ceschin et al., 2020). Duas estratégias utilizadas pelos algoritmos de classificação são: uma classe e multi classe. A seguir são apresentadas com mais detalhes cada uma dessas estratégias.

Na categoria de algoritmos de classificação uma classe os modelos são treinados apenas com amostras benignas. O objetivo desse modo de treinamento é desenvolver modelos capazes de identificar e definir os limites da classe de comportamento benigno. Esses algoritmos são projetados para serem aplicados em situações em que a classe negativa está ausente, possui poucas amostras ou não está bem definida (Khan e Madden, 2014).

Passando para outra abordagem, a categoria de algoritmos de classificação multi classe conta com amostras benignas e malignas para realizar os treinamento dos modelos ML. Nesse abordagem os modelos são treinados para avaliar amostras benignas e malignas e definir os limites e padrões característicos de cada classe. Esse modelo de treinamento pode ser implementado quando o número de amostras benignas e malignas são suficientes e possuem uma proporção equilibrada (Arp et al., 2014).

Em suma, as técnicas de ML para segurança proporcionam maneiras eficazes para a classificação de algoritmos por meio do treinamento de modelos que podem ser adaptados a diferentes contextos de abundância de dados, contribuindo para a detecção de *malwares* e para a segurança no meio digital.

3 REVISÃO BIBLIOGRÁFICA

Tratando-se de uma nova tecnologia para desenvolvimento de aplicações Web, a análise e detecção de comportamentos maliciosos em programas WebAssembly a fim de proteger o usuário é o principal motivador para o desenvolvimento do trabalho. Nessa seção serão discutidas as principais abordagens relacionadas a esse objetivo e os respectivos artigos com suas contribuições.

3.1 ANÁLISE ESTÁTICA EM WEBASSEMBLY

Por tratar-se de uma abordagem eficiente e segura para análise de comportamento de código, a análise estática foi empregada e diferentes trabalhos acadêmicos como forma de identificar vulnerabilidades em módulos WebAssembly.

Em (Quan et al., 2019) é proposta a ferramenta *EVulHunter* para análise estática de módulos WebAssembly para a plataforma de *blockchain* EOSIS, a fim de identificar principalmente vulnerabilidades do tipo falsa transferência. Por meio da construção e posterior análise do Grafo de Controle de Fluxo de um módulo binário, a ferramenta desenvolvida é capaz de identificar padrões pré-definidos de vulnerabilidades de maneira automática. Após realizar testes, (Quan et al., 2019) conclui que a aplicação desenvolvida possui uma acurácia total de 93,14% e precisão de 89,26%. Contudo reconhece que, por detectar apenas padrões constantes pré definidos, não é capaz de modelar e identificar outros tipos de vulnerabilidades que podem estar presentes nos módulos analisados.

No artigo (Stiévenart e De Roover, 2020) a ferramenta desenvolvida *Wassail* é a primeira abordagem de análise estática para WebAssembly, utilizando um método de análise composicional baseado em resumo. Nesse método as funções são analisadas individualmente, ignorando o contexto em que foram chamadas. Após a análise é inferido um resumo da função que posteriormente pode ser utilizado para outras funções que a chamam (Stiévenart e De Roover, 2020). Dessa maneira, O Wassail é capaz de focar no desenvolvimento e avaliação do fluxo da aplicação WebAssembly (Stiévenart e De Roover, 2020). Na conclusão dos testes é determinado que o *framework* desenvolvido possui 64% de precisão em sua avaliação.

Em (Brito et al., 2022) a abordagem utilizada consiste em gerar a partir de um arquivo binário o seu *Code Property Graph (CPG)* e realizar análises para produzir um relatório de vulnerabilidades. Nessa forma de avaliação do código, uma estrutura com múltiplas camadas é gerada para proporcionar uma visão mais rica e abrangente do código (Brito et al., 2022). Essa estrutura é chamada de CPG, e é uma combinação da representação da estrutura do código fonte, o grafo de controle de fluxo e o grafo de dependências do programa (Kim et al., 2022). Após as representações serem combinadas e o CPG gerado, consultas são realizadas ao grafo em busca de propriedades específicas que indicam a presença de vulnerabilidades no código, as consultas podem ser pré definidas ou recebidas como entrada para a avaliação (Brito et al., 2022). A avaliação dos experimentos realizados conclui que a ferramenta desenvolvida é capaz de detectar vulnerabilidades na interface entre o código binário WebAssembly e o JS, e é eficiente para detectar todas as vulnerabilidades de binários gerados a partir de códigos C e C++ (Kim et al., 2022).

3.2 OUTRAS ANÁLISES EM WEBASSEMBLY

Outras ferramentas para verificação de vulnerabilidades foram desenvolvidas utilizando diferentes procedimentos. A seguir iremos explorar alguns trabalhos desenvolvidos a partir de abordagens de análise dinâmica e híbrida, combinando elementos de estática e dinâmica.

No artigo (Szanto et al., 2018) é proposta a estratégia de *taint tracking* para a análise de fluxo de informações sensíveis durante a execução do módulo WebAssembly. Esse método de avaliação utiliza rótulos marcados como entradas de funções e possibilita a verificação de operações inseguras sem modificar a estrutura do código (Kim et al., 2022). Para realizar esse procedimento foi desenvolvido uma Máquina Virtual em JS para WebAssembly que permite o rastreamento de informações em código binário (Szanto et al., 2018). A avaliação realizada considerando custos de espaço e tempo (Szanto et al., 2018) conclui que a sobrecarga gerada é restrita linearmente e portanto aceitável, porém também aponta que possui funcionalidades faltantes como operações com ponto-flutuante.

(Fu et al., 2018) desenvolveu o *framework* chamado *TaintAssembly*. A ferramenta tem como estratégia de análise o *Taint Tracking*, com o enfoque de acompanhar a integridade do fluxo de dados durante a interação entre o módulo WebAssembly e o JS. A aplicação desenvolvida é uma versão modificada da *engine* JS V8 que utiliza todos os tipos de variáveis do WebAssembly como entradas marcadas para monitorar o comportamento do fluxo de dados. Após realizar testes de desempenho, (Fu et al., 2018) conclui que o *TainAssembly* possui uma sobrecarga razoável entre 5-12% em comparação ao interpretador padrão.

Partindo para uma abordagem híbrida de análise estática e dinâmica, (Lehmann e Pradel, 2019) desenvolve o *Wasabi*, um *framework* que permite realizar a modificação do módulo WebAssembly para implementar instrumentação binária e posteriormente examinar dinamicamente seu comportamento. O *Wasabi* apresentou sobrecargas entre 1.02x e 163x ao serem realizados testes de desempenho em diferentes programas e instruções. Com esses resultados (Lehmann e Pradel, 2019) conclui que a ferramenta possui um custo razoável para análises dinâmicas de grande porte.

Outro método para realizar análise dinâmica no código é através da utilização da técnica chamada *greybox fuzzer*. Por meio dela entradas são geradas automaticamente a fim de desencadear vulnerabilidades no programa sendo testado. No artigo (Lehmann et al., 2021) essa técnica é empregada por meio da aplicação desenvolvida chamada *Fuzzm*, que, juntamente com a implementação de canários na memória do módulo, tem como objetivo detectar *overflows* e *underflows* utilizando diretamente os módulos em binário. Na conclusão dos testes de desempenho (Lehmann et al., 2021) determina que a ferramenta *Fuzzm* gera uma pequena sobrecarga com a implementação do canário e, ao ser comparada com o AFL (*American fuzzy lop*), um *fuzzer* de software livre orientado a segurança (Fioraldi et al., 2020), foi concluído que a aplicação é 37% mais lenta.

3.3 MEDIDAS DE SEGURANÇA PARA WEBASSEMBLY

Variando de abordagens que implementam um melhor gerenciamento de memória, até técnicas para melhorar o isolamento do ambiente que o algoritmo irá rodar, diferentes procedimentos já foram propostos ao longo dos últimos anos para mitigar o risco de ataques ao usuário por meio da utilização de código WebAssembly malicioso. A seguir iremos explorar algumas das principais soluções propostas para aumentar a segurança em WebAssembly.

Atualmente a criptografia é amplamente utilizada para proteger informações sensíveis tanto de clientes quanto de servidores no ambiente Web. Uma quantidade significativa desses

algoritmos são implementados em JS por conta de seu suporte ao ecossistema Web. Contudo muitos questionamentos são levantados em relação à segurança de tais aplicações por conta de vulnerabilidades inerentes ao JS (Watt et al., 2019).

A fim de solucionar tais problemas (Watt et al., 2019) propõe o CT-Wasm, uma solução baseada em WebAssembly (Wasm) para implementar criptografia segura em aplicativos web. A ferramenta é implementada como um conjunto de extensões Wasm que podem ser usadas para implementar algoritmos criptográficos seguros e eficientes em tempo de execução. As extensões Wasm permitem a execução de códigos seguros em um ambiente web, garantindo a integridade, a confidencialidade e a autenticidade dos dados. (Watt et al., 2019) apresenta uma análise de segurança rigorosa da solução proposta, comparando-a com outras implementações de criptografia em aplicações web. Os resultados mostram que a solução proposta é eficiente e segura contra diferentes tipos de ataques cibernéticos, e pode ser usada em diferentes plataformas e navegadores web.

Erros de memória são alguns dos principais fatores de vulnerabilidade do WebAssembly. Por ser um formato de instruções que pode ser compilado por linguagens inseguras como C e C++, violações de segurança e privacidade são um grande risco para o usuário que deseja utilizar tais aplicações em seu navegador. Para ir de encontro a essas vulnerabilidades, (Michael et al., 2022) apresenta o *Memory-Safe WebAssembly*, solução para garantir a execução segura de código inseguro no ambiente WebAssembly. Essa ferramenta tem como objetivo fornecer uma garantia de memória segura, por meio da utilização de um compilador que traduz o código de baixo nível para um subconjunto seguro de instruções WebAssembly que não permite acesso não autorizado à memória. A solução proposta oferece garantias de segurança robustas com uma sobrecarga variando de 22%, quando apenas a segurança espacial de memória é implementada, à 198%, em que todas as seguranças de memória são implementadas, em diferentes programas de teste (Michael et al., 2022).

Outra solução relacionada à memória é proposta por (Kolosick et al., 2022). Pensando no mecanismo de segurança de isolamento de memória e em seu custo significativo em termos de desempenho, (Kolosick et al., 2022) propõem uma solução a fim de mitigar esses problemas, que permite transições quase sem custo entre o modo isolado e o modo não isolado do WebAssembly. A solução proposta consiste em uma abordagem que utiliza dois mecanismos complementares. O primeiro mecanismo é uma técnica de transição de contexto que permite ao WebAssembly ser executado em uma *sandbox* com controle de fluxo de informações. O segundo mecanismo é um conjunto de otimizações de código que minimizam a sobrecarga introduzida pela técnica de *sandboxing*. Os resultados mostraram que a abordagem proposta é capaz de fornecer um nível de segurança semelhante ao de outras implementações com um custo computacional significativamente menor. Além disso, os experimentos mostraram que a solução proposta é capaz de reduzir drasticamente o tempo de execução de programas em WebAssembly com *sandboxing* e controle de fluxo de informações, tornando a abordagem viável para aplicações em tempo real (Kolosick et al., 2022).

Devido à falta de verificação de tipos e limitações na implementação de verificação de segurança, vulnerabilidades como estouro de buffer, ponteiros inválidos e outros riscos relacionados à gestão de memória do WebAssembly são suscetíveis à exploração por agentes mal intencionados (Disselkoen et al., 2019). Baseada em técnicas de verificação de segurança progressiva, a proposta de (Disselkoen et al., 2019) consiste em uma série de verificações de segurança em tempo de execução que são executadas gradualmente em diferentes níveis de granularidade, desde o nível mais baixo (módulos) até o mais alto (funções). Essas verificações incluem análise de tipos, análise de fluxo de dados e análise de dependência de ponteiros. Os resultados dos testes realizados por (Disselkoen et al., 2019) mostram que a abordagem proposta

pode melhorar significativamente a segurança de memória no WebAssembly, sem impactar significativamente o desempenho da execução. Além disso, o estudo destaca a necessidade de uma abordagem progressiva para a verificação de segurança em WebAssembly, que possa equilibrar a segurança e o desempenho.

3.4 CONSIDERAÇÕES

Considerando os trabalhos que apresentam propostas de análise estática de WebAssembly (Seção 3.1) é possível perceber que diferentes abordagens foram utilizadas para aumentar a segurança em aplicações desenvolvidas. Contudo, ao passo que solucionam alguns problemas, cada artigo também possui suas limitações e elementos de análise que precisam ser considerados.

(Brito et al., 2022) apresenta a ferramenta Wasmati que usa análise de fluxo de dados e técnicas de análise de *taint* para identificar possíveis vulnerabilidades em módulos WebAssembly. O artigo mostra que a abordagem proposta é eficiente em encontrar vulnerabilidades em módulos WebAssembly em comparação com outras ferramentas existentes. No entanto, ela não é capaz de lidar com algumas estruturas de dados complexas em programas WebAssembly, como arrays multi-dimensionais e estruturas de dados definidas pelo usuário. Além disso, a ferramenta depende de outras ferramentas externas para realizar algumas análises, o que pode levar a uma diminuição da eficiência e precisão da análise.

(Romano e Wang, 2020) emprega uma abordagem que visa entender as características de aplicações WebAssembly por meio de técnicas de classificação. O artigo mostra que a ferramenta é eficiente em identificar as características dos aplicativos WebAssembly e pode ser usada para análise de segurança. Porém suas limitações são: a abordagem de classificação proposta, que se baseia em um conjunto limitado de recursos, como a contagem de instruções e a complexidade ciclomática, que podem não ser suficientes para identificar todos os tipos de vulnerabilidades presentes em um programa WebAssembly. E a dependência de um conjunto pré-definido de classes de vulnerabilidades, o que pode limitar a capacidade de identificar novos tipos de vulnerabilidades.

(Quan et al., 2019) apresenta uma ferramenta que visa encontrar vulnerabilidades específicas em contratos inteligentes EOSIO que podem ser explorados usando WebAssembly. Suas limitações são a dependência da existência de uma função específica no código fonte que represente uma transferência, o que pode limitar sua capacidade da ferramenta de detectar vulnerabilidades em casos em que as transferências são realizadas de maneira não convencional, e o fato de que a ferramenta se concentra especificamente em uma classe de vulnerabilidades (vulnerabilidades de transferência falsa) e pode não ser eficaz na detecção de outras classes de vulnerabilidades em programas WebAssembly.

Passando para a avaliação de outras abordagens que foram utilizadas para análise dos módulos WebAssembly (Seção 3.2), iremos identificar a seguir os fatores positivos e limitantes relacionados a cada abordagem.

O artigo (Szanto et al., 2018) apresenta uma abordagem para rastreamento de contaminação de dados em WebAssembly. A proposta é utilizar um instrumentador que adiciona marcas de contaminação nas variáveis, possibilitando a detecção de pontos vulneráveis no código. Ao final do artigo é apresentado um estudo de caso que demonstra sua eficácia na detecção de vulnerabilidades em códigos de exemplo. No entanto, o estudo de caso apresentado é limitado e não considera a escalabilidade da abordagem para códigos maiores e mais complexos. Além disso, a técnica de rastreamento de *taint* pode afetar o desempenho da execução do código, tornando-a inviável para sistemas críticos.

Passando para o artigo (Fu et al., 2018), é proposta uma técnica de controle de fluxo de informação baseada em contaminação de dados. A abordagem consiste em aplicar *taint tracking* nas variáveis, utilizando uma técnica de análise de fluxo de controle de informação. A solução mostrou-se eficiente em detectar vulnerabilidades em estudos de caso, porém possui limitações na dependência de uma ferramenta externa para realizar a análise de *taint*, que pode gerar um *overhead* significativo de desempenho, e a técnica proposta não é capaz de lidar com instruções de salto condicional, o que limita sua eficácia em programas mais complexos.

No artigo (Lehmann et al., 2021) podemos analisar a abordagem de *fuzzing* para detecção de problemas de memória em códigos WebAssembly. A técnica proposta utiliza uma ferramenta de *fuzzing* de dados para realizar alterações em módulos WebAssembly e um instrumentador para coletar informações sobre o comportamento do programa. A solução foi avaliada em um conjunto de programas de teste e apresentou resultados promissores. Porém suas limitações são: a dependência de uma base de dados de tipos de vulnerabilidades conhecidas, o que pode limitar a capacidade da técnica de identificar novas vulnerabilidades, e a dependência de uma infraestrutura de testes automatizados, que pode exigir recursos computacionais significativos.

Por fim tempos o artigo (Lehmann e Pradel, 2019), que apresenta uma estrutura para análise dinâmica de códigos WebAssembly. A abordagem proposta é capaz de monitorar a execução do programa e identificar possíveis vulnerabilidades em tempo de execução. A solução foi avaliada em estudos de caso e mostrou-se eficiente em detectar vulnerabilidades em módulos WebAssembly, no entanto suas limitações são a necessidade de modificar o ambiente de execução para suportar a análise dinâmica, o que pode interferir na execução do programa original e afetar a precisão dos resultados, e a falta da capacidade para lidar com programas maliciosos que tentam evadir a análise, como por meio do uso de técnicas *anti-debugging*.

Por fim, com o propósito de determinar as contribuições e pontos em aberto relacionados à segurança em WebAssembly (Seção 3.3), iremos avaliar as limitações dos artigos apresentados na subseção de medidas de segurança para WebAssembly.

O primeiro artigo, (Watt et al., 2019), propõe uma abordagem de criptografia segura, na qual a solução desenvolvida é um conjunto de extensões Wasm que possibilita a implementação de algoritmos criptográficos. Embora o artigo apresente uma solução promissora, as seguintes limitações podem ser observadas: questões de performance e de compatibilidade com outras linguagens de programação que utilizam o WebAssembly não são abordadas pelo artigo; além disso, a solução proposta é limitada a um conjunto específico de algoritmos criptográficos, não sendo capaz de garantir segurança para todos os tipos de criptografia.

O segundo artigo, (Michael et al., 2022), propõe uma abordagem para executar código não seguro em um ambiente seguro, através da utilização de técnicas de verificação de memória e análise de fluxo de controle. A limitação deste artigo é que a solução proposta não considera a segurança em relação a outros aspectos, como a segurança da rede, prevenção de ataques de injeção de código e de engenharia reversa. Além disso, a solução não considera a possibilidade de vulnerabilidades em bibliotecas de terceiros, o que pode comprometer a segurança do sistema como um todo.

O terceiro artigo, (Kolosick et al., 2022), propõe uma solução por meio do isolamento de componentes potencialmente inseguros em ambientes protegidos, sem custo computacional significativo. Embora a abordagem proposta proteja o código *host* da execução de código potencialmente inseguro, ela não oferece proteção contra vulnerabilidades no próprio *host*. Além disso, ela não aborda a questão de garantir que o código em si seja seguro ou livre de vulnerabilidades. Portanto, a solução proposta é uma ferramenta complementar à análise de segurança de código para garantir a execução segura de aplicativos Wasm.

Por fim, o quarto artigo, (Disselkoen et al., 2019), propõe uma abordagem para garantir a segurança da memória progressivamente. A proposta atualmente só é capaz de proteger contra vulnerabilidades de uso após liberação (*use-after-free*) e *buffer overflow*, deixando outras vulnerabilidades de memória sem proteção. Além disso, a abordagem ainda apresenta alguns desafios em relação à compatibilidade com a especificação Wasm e com outras soluções de segurança já existentes.

A Tabela 3.1 apresenta o levantamento dos artigos relacionados à analise de código WebAssembly. Apresentando o artigos com a respectiva técnica de análise empregada pela ferramenta desenvolvida e a disponibilidade do conjunto de dados utilizado nos testes. É notável que nenhum dos artigos disponibiliza de forma pública o conjunto de dados utilizado para avaliação.

Artigo	**Tipo de avaliação**	**Dataset**
(Brito et al., 2022)	Análise estática de vulnerabilidades	Não diponível
(Stiévenart e De Roover, 2020)	Análise estática de vulnerabilidades	Possui código aberto
(Romano e Wang, 2020)	Classificação de aplicativos	Não disponível
(Quan et al., 2019)	Análise de vulnerabilidades	Não disponível
(Szanto et al., 2018)	Rastreamento de tinta	Não disponível
(Fu et al., 2018)	Controle de fluxo de informações baseado em tinta	Não disponível
(Lehmann et al., 2021)	Detecção de erros de memória	Não disponível
(Lehmann e Pradel, 2019)	Análise dinâmica	Não disponível

Tabela 3.1: Trabalhos Relacionados

Os artigos analisados demonstram que existem diversas técnicas de análise estática e dinâmica que visam identificar vulnerabilidades e classificar aplicativos WebAssembly. No entanto, há um ponto em aberto em relação à identificação das funções individuais presentes em módulos WebAssembly. A falta de uma abordagem eficiente para essa tarefa dificulta a análise de segurança, uma vez que não é possível identificar a funcionalidade específica que pode apresentar falhas de segurança. Estas estratégias não focam em explorar informações chaves encontradas nos binários WebAssembly, que por consequência podem ser utilizadas para a caracterização das respectivas aplicações.

Dessa forma, uma estratégia voltada na classificação de binários através de modelos de aprendizado de maquina pode possibilitar a identificação automatica de funções presentes em módulos WebAssembly e classificá-las em categorias relevantes para a análise de segurança, como entrada de dados, manipulação de arquivos, acesso à rede, entre outras. Com essa abordagem, seria possível facilitar a identificação de vulnerabilidades e aumentar a eficácia das técnicas de análise estática e dinâmica.

4 PROPOSTA

Nesta seção serão discutidas as principais questões relacionadas à problemática que este artigo visa atingir, a estratégia utilizada para alcançar este objetivo, a coleta de dados que foi realizada e os métodos de avaliação empregados para avaliar os resultados da solução proposta.

4.1 PROBLEMÁTICA

Como evidenciado no Capítulo 2 o formato de instruções binárias WebAssembly possui grande foco em desempenho e segurança. Por esses motivos, e por sua característica de ser executado em um ambiente virtualizado, módulos em WebAssembly têm sido cada vez mais explorados como alternativa a outras opções de linguagem de desenvolvimento de aplicações web, bem como aplicações em *blockchain* para a criação de *smart contracts*. Contudo, embora o WebAssembly tenha sido projetado para fornecer segurança e *sandboxing*, a plataforma ainda está suscetível a vulnerabilidades de segurança explorada por atacantes ou até mesmo o desenvolvimento de aplicações maliciosas.

Como apresentado no Capítulo 3, diferentes questionamentos são levantados em relação à real segurança que o WebAssembly é capaz de garantir às suas aplicações. Diferentes vulnerabilidades relacionadas ao gerenciamento de memória de módulos WebAssembly são apresentadas (Michael et al., 2022; Kolosick et al., 2022; Disselkoen et al., 2019; Lehmann et al., 2021). Problemas como estouro da pilha podem ser explorados para realizar a execução de código malicioso ou para obter acesso não autorizado ao sistema, levando a sérios riscos em relação à confidencialidade e integridade da aplicação.

Vulnerabilidades relacionadas ao fluxo de dados e utilização de funções inseguras também são preocupações expostas (Szanto et al., 2018; Fu et al., 2018). Por meio da utilização de códigos de desvio de fluxo, códigos maliciosos podem ser executados no sistema. Por meio de manipulação do fluxo de execução e por meio de técnicas de injeção de dependência, aplicações WebAssembly podem ser alvo de ataques que tem como objetivo a execução de código malicioso, trazendo riscos ao usuário.

Com esses problemas em mente, e tendo em vista a crescente adoção do WebAssembly em diversas aplicações, torna-se cada vez mais importante identificar vulnerabilidades de segurança em tempo hábil, a fim de tomar medidas corretivas. Nesse sentido, a análise estática de código apresenta-se como uma solução atrativa para a identificação de vulnerabilidades de segurança antes da execução do código. Contudo, como apresentado na Seção 3.4, soluções como as (Brito et al., 2022; Stiévenart e De Roover, 2020; Quan et al., 2019) possuem limitações como incapacidade de lidar com estruturas de dados complexas, capacidade limitada de identificar novos tipos de ataques ou um escopo muito limitado em relação à sua análise.

Diante disso, a utilização de modelos de aprendizado de máquina se apresenta como uma solução promissora para automatizar a análise estática de binários WebAssembly. Essas técnicas envolvem a utilização de algoritmos de aprendizado de máquina para identificar padrões em grandes conjuntos de dados e gerar modelos. Esses modelos podem ser usados para identificar vulnerabilidades de segurança em códigos WebAssembly de forma eficiente e eficaz, podendo também ajudar na identificar vulnerabilidades de segurança que são difíceis de detectar por meio de análise estática tradicional, o que aumenta a eficácia da detecção de vulnerabilidades e, consequentemente, a segurança do WebAssembly.

4.2 ESTRATÉGIA

O foco desse trabalho é a análise estática de binários WebAssembly utilizando técnicas de aprendizado de máquina para a classificação dos mesmos. Primeiramente foi avaliado o potencial do uso de características extraídas de binários WebAssembly para a classificação, e posteriormente para a detecção de intrusão. Para isso foi utilizado o formato de depuração *Debugging With Attributed Record Formats* (DWARF) para extrair características dos códigos binários WebAssembly. O DWARF permite que sejam acessadas informações importantes sobre programas em WebAssembly, como nomes de funções e variáveis, tipos de dados e localização de código fonte (Delendik, 2020).

Através deste recurso, características dos binários WebAssembly podem ser acessadas e extraídas, permitindo uma avaliação estática dos binários. Estas informações foram filtradas e avaliadas para o processo de classificação e detecção de intrusão.

Por meio dessas informações modelos de aprendizado de máquina foram treinados para detecção de anomalias presentes nos binários WebAssembly. Foi empregado o uso de aprendizado de máquina a fim de automatizar o processo de análise e classificação, e foi determinada a viabilidade da utilização dessa classificação para detecção de intrusão. Ao utilizar um modelo de aprendizado de máquina, o risco de erro humano é minimizado e a utilização desse método permitiu a detecção de vulnerabilidades que não são facilmente identificadas por técnicas de análise estática tradicionais, ajudando a melhorar a segurança do código em geral.

Inicialmente foi realizada a avaliação das informações extraídas por meio do DWARF e determinou-se que elas são suficientes para a extração de características para a identificação de anomalias no código binário WebAssembly. Foi então realizada a avaliação dos modelos de aprendizado de máquina, para identificar a capacidade de aprendizado que os dados extraídos dos binários possuem, e os resultados obtidos.

4.3 COLETA DE DADOS

Para a coleta de dados foram utilizados dois datasets com amostras benignas e malignas para compor o conjunto de dados utilizado para treinamento e avaliação dos modelos ML. O conjunto de amostras benignas é composto por 127 módulos binários WebAssembly selecionado a partir do dataset proposto por (Lehmann e Pradel, 2022). Para compor o conjunto de amostras malignas foram selecionados 150 binários a partir do dataset proposto por (Stiévenart et al., 2022).

4.4 MÉTODOS DE AVALIAÇÃO

A aplicação de técnicas de aprendizado de máquina na área de segurança cibernética tem sido amplamente estudada na literatura, devido à sua eficiência em detectar anomalias e prever ataques. Modelos de aprendizagem são capazes de analisar grandes volumes de dados e identificar padrões que seriam imperceptíveis para os seres humanos, permitindo a detecção de anomalias de forma mais rápida e com maior precisão (Shaukat et al., 2020).

Por esses motivos, foram exploradas técnicas de aprendizado de máquina com o objetivo em classificar adequadamente os binários WebAssembly. Através desta estratégia, foi possível avaliar o impacto do uso das informações extraídas dos binários para a detecção de intrusão. Para isso foram realizados treinamentos e posteriormente testes com classes de códigos maliciosos e não maliciosos. A solução também foi avaliada em relação a outras técnicas de análise estática de códigos binário.

5 AVALIAÇÃO DE RESULTADOS

Nesta seção são apresentados e discutidos os principais resultados obtidos por meio da implementação da Estratégia proposta no Capítulo 4. As Seções 5.1, 5.2 e 5.3 descrevem os atributos e características selecionados para caracterizar os binários, a coleta e a preparação dos dados utilizados nos experimentos, respectivamente. Em seguida, na Seção 5.4, são apresentadas analisadas e interpretadas as principais métricas de desempenho alcançadas, destacando as contribuições proporcionadas pela solução proposta e discutindo suas limitações. Por fim, na Seção 5.5, são apresentadas as principais conclusões que podem ser extraídas dos resultados, comparando-as com trabalhos relacionados e discutindo limitações e possíveis melhorias.

5.1 SELEÇÃO DE ATRIBUTOS

Para obter as informações dos binários WebAssembly foi utilizado o DWARF. Sendo um formato de arquivo e um conjunto de convenções para representar informações de depuração de programas, o DWARF é capaz de fornecer informações relevantes para análise estática a respeito do binários WebAssembly, como por exemplo informações de tipos de variáveis, declarações de funções, linguagem da qual o programa foi compilado, compilador utilizado, entre outros.

O DWARF tem como objetivo definir uma representação de baixo nível de um programa fonte a fim de auxiliar no processo de depuração do código. Para isso, durante o processo de compilação são geradas *tags* e atributos específicos com base no código fonte. Cada *tag* fornece uma descrição de uma entidade correspondente no programa fonte, sendo por meio da análise dessas descrições que os algoritmos de ML são capazes de realizar classificações relacionadas ao comportamento do programas binários.

A abordagem utilizada para caracterização dos arquivos foi o cálculo de frequência de características específicas de cada binário WebAssembly. Para isso foram extraídas 15 características das seções de depuração DWARF que podem ser classificadas em 5 grupos: informações básicas, informações de tipos de dados, informações de sub-rotinas e subprogramas, informação de estruturas em memória e informações de compartilhamento externo. A seguir serão exploradas as informações extraídas relacionadas a cada grupo de características.

5.1.1 Informações básicas

No grupo de informações básicas estão presentes as características de tamanho do programa em número de linhas, o número de rótulos presentes no código e a linguagem do código fonte.

A característica de tamanho do programa foi extraída pois o número de linhas pode ser um indicativo de tentativa de ofuscação do código malicioso por meio da inserção de linhas irrelevantes. O número de rótulos foi extraído por meio da contagem do número de *tags* `DW_TAG_label` presentes no arquivo DWARF.Um rótulo é uma maneira de identificar um local em código e normalmente é utilizado como alvo de instruções de desvio de fluxo de controle. A informação a respeito do número de rótulos é relevante pois pode indicar a complexidade de estruturas de controle presentes no programa e a tentativa de ofuscação em relação a classificadores que utilizam informações de fluxo de controle para realizar sua análise. A característica de linguagem do código fonte do binário WebAssembly foi extraída por meio do atributo `DW_AT_language`. Essa informação foi utilizada pois ao realizar a compilação

de códigos para o formato de instruções WebAssemby, diferentes vulnerabilidades podem ser importadas de linguagens específicas para o código binário.

5.1.2 Informações de tipos de dados

O conjunto de informações sobre tipos de dados inclui o número de declarações de tipos, tipos inteiros declarados, tipos inteiros sem sinal declarados, tipos palavra declarados e tipos ponteiro para arquivo declarados. Como o WebAssembly tem um conjunto restrito de tipos de dados nativo, é necessária a definição esses tipos nos módulos WebAssembly.

O cálculo de frequência de declarações de tipos de dados foi realizado por meio da contagem de *tags* `DW_TAG_typedef` presentes no código. Essa informação é importante para a caracterização dos binários pois, quando avaliada juntamente com o número de declarações de cada tipo, pode indicar os tipos de dados predominantes sendo utilizados no programa.

Para determinar o número de declarações de cada tipo foi realizado o cálculo de frequência de valores do atributo `DW_AT_type`. Esse atributo pertence a *tag* `DW_TAG_typedef` e seu valor é uma referência para o tipo nomeado pela definição de tipo no código. Para realizar a caracterização, os principais nomes de tipos foram agrupados em categorias. No conjunto de declarações de tipo inteiro foram considerados os nomes: *int, long_int, long_long_int* e *wint_t*. No conjunto de declarações de tipo inteiro sem sinal foram considerados os nomes: *unsigned_int, uint8_t, uint32_t, uint64_t, long_unsigned_int* e *long_long_unsigned_int*. No conjunto de declarações de tipo palavra foram considerados os nomes: *const_char*, restrict_const_char*, char*, restrict_const_wchar_t*, wchar_t*, const_wchar_t*, restrict_char*, basic_string<char>, std::char_traits<char>, std::allocator<char> *, restrict_wchar_t** e *restrict_wchar_t***. No conjunto de declarações de tipo ponteiro para arquivo foram considerados os nomes: *FILE*, restrict_FILE*, FileMetaData** e *FileMetaData***.

Além de deu uso comum em operações aritméticas, o uso de inteiros em módulos WebAssembly também está relacionado com a comunicação com aplicações externas e controle de fluxo do programa. Considerando *malwares* que utilizam os diferentes tipos de inteiros para forjar comunicações ou para introduzir padrões anômalos de fluxo de dados, o número de declarações de tipos inteiros foi considerado para realizar a caracterização dos binários WebAssembly. Os tipos inteiros sem sinal também foram considerados para a caracterização pois seu uso está muito relacionado com o acesso à memória por meio de índices, o que pode ser usado para explorar vulnerabilidades de memória por meio de *overflows* ou *underflows*.

Em códigos maliciosos frenquentemente é implementado o uso de strings para ofuscação de código e injeção de código maliciosos. Considerando esses fatores, foi realizado o cálculo de frequência de declarações desse tipo a fim de fornecer informações que podem indicar a utilização maliciosa de *strings*. O uso do número de declarações de tipo ponteiro para arquivo foi considerado para a caracterização dos binários com o objetivo de fornecer informações que podem indicar a manipulação de arquivos de forma maliciosa, seja extraindo dados de arquivos ou inserindo informações privadas do usuário.

5.1.3 Informações de sub-rotinas e subprogramas

No grupo de informações de sub-rotinas e subprogramas, foi considerado o número de variáveis declaradas, o número de declarações de sub-rotinas *inline*, o número de declarações de subprogramas e o número de parâmetros desses subprogramas.

Para determinar o número de variáveis declaradas no programa foi utilizada a *tag* `DW_TAG_variable`. Essa característica foi utilizada para caracterizar os binários pois o uso de variáveis está relacionado com a definição dos tipos vistos na Subseção 5.1.2 e o

número de declarações de variáveis pode fornecer informações sobre a complexidade a estrutura do código-fonte. Para extrair o número de declarações de sub-rotinas *inline* foi utilizada a *tag* `DW_TAG_inlined_subroutine`. Essa característica foi escolhida para realizar a caracterização dos binários pois fornece informações sobre a complexidade do código e a presença de um grande número de sub-rotinas *inline* pode indicar uma tentativa de ocultar o comportamento malicioso. O número de declarações de subprogramas e o número de declarações de parâmetros foram extraídos dos binários por meio do cálculo de frequência das *tags* `DW_TAG_subprogram` e `DW_TAG_formal_parameter`, respectivamente. A utilização desses números para caracterização dos binários pode oferecer informações a respeito da complexidade do código.

5.1.4 Informação de estruturas em memória

A informação de estruturas em memória é representada pelo cálculo de frequência de membros declarados quando há o uso de estruturas ou classes. Foi utilizada a *tag* `DW_TAG_member` para definir o número de membros declarados e essa informação faz parte da caracterização dos binários WebAssembly pois pode fornecer informações a respeito da complexidade da estrutura de dados do programa.

5.1.5 Informações de compartilhamento externo

Por não possuir uma biblioteca padrão, os módulos WebAssembly realizam interações com o ambiente externo por meio da importação de funções externas e *flags* de controle. O grupo de informações de compartilhamento externo inclui dados sobre a quantidade de atributos que determinam se as sub-rotinas são parte de um programa externo ou produzem informações acessíveis externamente e a quantidade de *tags* e atributos relacionados a utilização de chamadas de funções GNU.

Para determinar o número de sub-rotinas que são importadas de fontes externas ou que produzem informações que podem ser acessadas externamente foi calculada a quantidade de atributos `DW_AT_external`. Essas informações são relevante para a caracterização dos binários pois, se for identificado um número anômalo, podem indicar o uso para ofuscação de chamadas externas de funções maliciosas. Para determinar o número de chamadas de funções GNU e o número de entradas em subprogramas GNU foram utilizadas a *tag* `DW_TAG_GNU_call_site` e o atributo `DW_AT_GNU_all_call_site`, respectivamente. O uso dessas informações foi considerado para a caracterização dos binários WebAssembly pois são indicativos do ambiente em que os códigos foram compilados e podem oferecer indicativos a respeito da complexidade de fluxo de controle do binário.

5.2 EXTRAÇÃO DE DADOS

A extração de dados foi realizada em três etapas. Primeiramente foi realizada a extração das informações de depuração DWARF dos arquivos binários WebAssembly por meio da ferramenta LLVM (*Low Level Virtual Machine*) (Lattner e Adve, 2004), gerando os respectivos arquivos DWARF. Após a extração e geração dos arquivos DWARF foi então realizada a geração do arquivo CSV (*Comma-Separated Values*) contendo as características de todos os arquivos binários analisados. Por fim foi realizado o preprocessamento no arquivo CSV a fim de preparar os dados para servirem de entrada para os algoritmos ML. A seguir iremos explorar mais detalhadamente os conceitos e processos envolvidos em cada uma dessas etapas.

O LLVM é uma infraestrutura de compilador de código aberto que oferece suporte para a análise e transformação de programas arbitrários (Lattner e Adve, 2004). Dentre suas funcionalidades, o LLVM oferece suporte para a geração da seção de depuração DWARF no processo de compilação de códigos para o formato de instruções WebAssembly, portanto ele é a principal ferramenta utilizada por diferentes compiladores para realizar essa tarefa.

A Figura 5.1 apresenta o processo de geração de uma aplicação WebAssembly com as informações DWARF. O processo é composto de três etapas, sendo (1) Compilação do código-fonte da linguagem de programação para o formato de instruções WebAssembly; (2) Geração da seção de depuração DWARF por meio de flags específicas durante o processo de compilação, para que sejam produzidas as informações de depuração; (3) Associação da seção de depuração em formato DWARF gerada com o resultado da compilação do código-fonte em um binário executável.

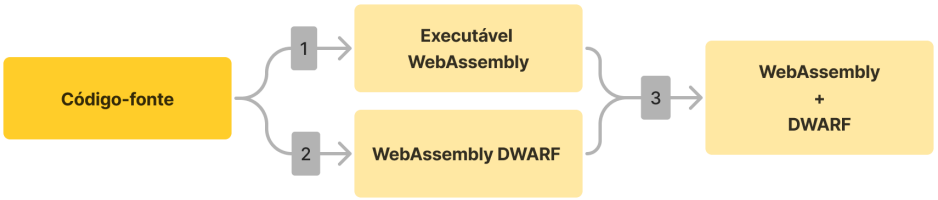


Figura 5.1: Exemplo de compilação e geração da seção DWARF para arquivo WebAssembly

Uma vez associados em um único arquivo executável, a extração da seção de depuração DWARF de binários WebAssembly é outra funcionalidade oferecida pelo LLVM. Utilizando essa funcionalidade foram então extraídas as informações DWARF de cada arquivo binário e passou-se para a segunda etapa do processo de Extração de dados, a geração do arquivo CSV de entrada para os algoritmos de ML.

Na fase de geração do arquivo CSV todas as características foram extraídas dos arquivos DWARF por meio da extração de características e cálculo de frequências descritos na Seção 5.1. Os valores inteiros e a característica de linguagem extraída referentes a caracterização dos arquivos foram inseridas em uma mesma tabela, juntamente com a respectiva definição booleana se o arquivo é malicioso ou não.

Finalmente, com o arquivo CSV contendo todas as características de todos os arquivos binários WebAssembly, foi realizada a etapa de preprocessamento para preparar os dados para os algoritmos ML. Primeiramente os dados foram separados em duas tabelas, uma contendo as informações de todos os arquivos para servir de entrada para treinamento dos algoritmos multi classe e outra contendo apenas as informações referentes aos arquivos benignos para servir de entrada para treinamento dos algoritmos com uma classe. Após isso as colunas de identificação do arquivo e de determinação de maliciosidade foram retiradas das tabelas, e então foi realizada a transformação da característica de linguagem do código fonte para um valor inteiro representando cada classe de linguagens das duas tabelas. Ambas as tabelas passaram pelo processo de normalização L1 e então foram divididas entre tabelas de treino e teste com tamanho 50%.

Ao fim da última etapa os dados foram devidamente extraídos dos arquivos binários, lidos e inseridos no arquivo CSV, preprocessados e separados entre treino e teste e prontos para serem utilizados nos modelos ML com a configuração padrão para classificação entre arquivos de comportamento benigno ou malicioso.

5.3 DATASET PROPOSTO

O Dataset utilizado para extração de dados é composto de 127 amostras benignas e 150 amostras malignas de binários WebAssembly. As amostras benignas foram selecionadas do dataset proposto pelo artigo (Lehmann e Pradel, 2022). Esse conjunto de dados é composto por mais de 4 mil arquivos objeto com código WebAssembly e seção de depuração DWARF que foram compilados de arquivos de pacotes fonte do Ubuntu escritos em C e C++. Durante o processo de seleção das amostras foram considerados dois critérios: selecionar binários buscando alcançar maior variabilidade de tamanho e propósito. Portanto foram selecionados arquivos que compõem diferentes ferramentas e bibliotecas. Para as amostras malignas foram selecionados binários do dataset proposto pelo artigo (Stiévenart et al., 2022). Com o objetivo de contemplar uma maior variedade dos tipos de vulnerabilidades exploradas pelos códigos maliciosos, foram escolhidos binários de diferentes categorias dentro do dataset.

5.4 RESULTADOS

Para a realização dos testes foram utilizadas as configurações padrão dos algoritmos. A seguir, na Tabela 5.1, estão descritos os resultados obtidos pelos algoritmos de ML. Para os testes foram utilizados algoritmos multi classe e uma classe. Esta abordagem foi adotada para determinar se os modelos são capazes de classificar corretamente utilizando apenas informações de binários benignos para treinamento, o que é refletido nos resultados dos algoritmos uma classe, ou se é necessária a utilização da caracterização de binários malignos para o treinamento dos modelos ML, refletido nos resultados dos algoritmos multi classe. É possível perceber que os resultados variam de acordo com a estratégia de classificação empregada por cada algoritmo, porém é importante destacar que em quatro algoritmos multi classe os resultados se apresentam favoráveis para serem utilizados como uma estratégia de detecção de anomalias.

Tipo	**Classifier**	**Precision**	**Recall**	**F1Score**	**Accuracy**	**BAC**	**Brier Score**
Multi classe	AdaBoost	96,10%	100%	98,01%	97,84%	97,69%	2,16%
	K-Nearest Neighbors	93,67%	100%	96,73%	96,40%	96,15%	3,6%
	Multi-Layer Perceptron	71,15%	100%	83,15%	78,42%	76,92%	21,58%
	RandomForest	98,67%	100%	99,33%	99,28%	99,23%	0,72%
	XGBoost	96,10%	100%	98,01%	97,84%	97,69%	2,16%
Uma classe	Isolation Forest	90,95%	89,21%	88,93%	89,21%	87,70%	-
	Support Vector Machines	84,79%	79,14%	77,52%	79,14%	76,23%	-

Tabela 5.1: Desempenho dos algoritmos de ML para a detecção de anomalias.

Os classificadores que fazem parte do subgrupo multi classe são os algoritmos que utilizam as duas classes de arquivos, benignos e maliciosos, para realizar seu treinamento. Avaliando inicialmente a coluna *Precision* é possível notar que os algoritmos *K-Nearest Neighbors* e *Multi-Layer Perceptron* foram os modelos mais afetados pela classificação de falsos

positivos, classificação de binários benignos como maliciosos, com valores de 93,67% e 71,15% respectivamente, porém os outros modelos não sofreram com esse problema, alcançando valores maiores que 96%. Atentando-se para a coluna *Recall*, é notável que nenhum dos modelos foi impactado por falsos negativos durante a fase de testes, atingindo o valor de 100%. A avaliação de forma conjunta o impacto de falsos positivos e falsos negativos é refletida na coluna *F1Score*, nela podemos perceber também o impacto da classificação de falsos positivos nos algoritmos *K-Nearest Neighbors* e *Multi-Layer Perceptron*, que atingiram os valores mais baixos de 96,73% e 83,15%, os outros modelos que foram menos afetados no valor de *Precision* atingiram valores acima de 98%, com destaque para o modelo *random forest* que atingiu a marca de 99,33%.

Passando para a coluna *Accuracy* é notável a capacidade de aprendizagem dos padrões por parte dos modelos, avaliando o número de classificações realizadas corretamente em relação ao número total de classificações realizadas os modelos atingiram valores acima de 96%, com excessão apenas para o classificador *Multi-Layer Perceptron*, que obteve o valor aproximado de 78%. Temos por fim as colunas *BAC* (*Balanced Acuracy*) e *Brier Score*, em que o único destaque negativo é o modelo *Multi-Layer Perceptron*. Seu valor de 21,58% para o *Brier Score* demonstra que o modelo não está balanceado corretamente para realizar a classificação de anomalias. Apesar disso, é notável que os demais modelos demonstram que estão devidamente balanceados com valores acima de 96% na coluna *BAC* e a baixo de 4% na clona *Brier Score*.

Ao realizar a análise dos classificadores multi classe é possível perceber que o modelo que mais se destaca em seus resultados é o *RandomForest* seguido do *AdaBoost* e *XGBoost* com bons resultados. Os modelos que foram menos eficientes em sua classificação foram então o *K-Nearest Neighbors* e, por fim, o *Multi-Layer Perceptron*.

Passando para o segundo subgrupo denominado Uma classe, os classificadores que o compõem foram treinados com apenas as amostras benignas, tendo a função, portanto, de detectar padrões que fogem do que foi estabelecido durante a fase de treinamento. Analisando o impacto de amostras benignas que foram classificadas como malignas pelos modelos, representado pela coluna *Precision*, o impacto de falsos positivos é claro com os modelos atingindo valores aproximados de 85% e 91%, apesar de ainda assim terem sido obtidos valores maiores do que o *Multi-Layer Perceptron*. É notável também que, diferentemente dos algoritmos Multi classe, os classificadores *IsolationForest* e *Support Vector Machines* foram impactados pela classificação de falsos negativos, o que pode ser observado pela coluna *Recall* com os valores de 89,21% e 79,14%, respectivamente. De forma combinada, falsos positivos e falsos negativos compõem os valores da coluna *F1Score* que foram abaixo do que pode ser considerado aceitável, com valores aproximados de 88% e 77%. Avaliando a coluna *Accuracy* também nota-se a diferença de aprendizado para os casos positivos e negativos verdadeiros, com valores de 89,21% e 79,14%, abaixo dos alcançados pelos modelos multi classe, mas ainda assim acima do que o *Multi-Layer Perceptron*. A coluna *BAC* também demonstra o desbalanceamento dos algoritmos para classificação de binários maliciosos, com o *Isolation Forest* atingindo o valor de 87,7% e o modelo *Support Vector Machines* 76,23%.

De maneira geral, com exceção dos modelos *IsolationForest*, *Support Vector Machines* e *Multi-Layer Perceptron*, os algoritmos de ML alcançaram valores adequados para classificação de binários WebAssembly. Portanto, é possível afirmar que a detecção de anomalias utilizando modelos de ML por meio de características extraídas dos arquivos binários WebAssembly é uma área de pesquisa que possui potencial e pode ser mais explorada a fim de produzir modelos mais robustos.

5.5 DISCUSSÃO

O uso de técnicas de aprendizado de máquina aplicadas à segurança cibernética é uma área promissora e com eficácia comprovada para a detecção de anomalias e algoritmos maliciosos. Como demonstrado na sessão de Resultados (5.4), o uso de modelos ML para detecção de anomalias em binários WebAssembly apresenta resultados favoráveis e se mostra como uma área promissora a ser explorada. Por meio da extração de características dos binários WebAssembly e o uso de algoritmos ML para análise e classificação em grupos de códigos benignos ou maliciosos, foram obtidos resultados com desempenhos (coluna *F1Score*) de 96% à 99% para mais da metade dos algoritmos avaliados, e *Accuracy* com valores acima de 96% para quatro dos sete modelos explorados na abordagem. Com base nesses resultados é possível afirmar que a aplicação de modelos ML em características de binários WebAssembly para a detecção de código malicioso nos módulos é uma área promissora que pode ser utilizada para proteger sistemas contra intrusão.

Na Seção 3.4 foram apresentadas três soluções (Brito et al., 2022; Stiévenart e De Roover, 2020; Quan et al., 2019) para análise estática e detecção de vulnerabilidades em códigos WebAssembly e foram elencadas as limitações existentes em cada abordagem. Com base nas limitações estabelecidas anteriormente, a seguir será apresentado como a abordagem proposta supera essas limitações e avança na área análise estática para detecção de vulnerabilidades em códigos WebAssembly.

A solução proposta neste estudo supera a limitação da abordagem apresentada pela ferramenta *EVulHunter* (Quan et al., 2019), que foca principalmente na identificação de vulnerabilidades do tipo falsa transferência em módulos WebAssembly para a plataforma de *blockchain* EOSIS. A aplicação dos modelos de aprendizado de máquina permite uma detecção mais abrangente e flexível de código malicioso nos módulos WebAssembly, pois não se limita apenas a padrões pré-definidos de vulnerabilidades. Os resultados obtidos nesta pesquisa mostram a eficácia da solução proposta em detectar uma variedade de anomalias nos binários WebAssembly. Portanto, a solução proposta oferece uma abordagem mais abrangente em relação ao escopo de identificação de vulnerabilidades e flexível, podendo ser aplicada em diversos contextos, ajudando a proteger sistemas contra intrusões e vulnerabilidades.

Em relação à abordagem apresentada pela ferramenta *Wassail* em (Stiévenart e De Roover, 2020), que utiliza análise estática e um método baseado em resumo para análise de módulos WebAssembly, a solução proposta neste estudo se destaca em seus resultados de classificação. Ao passo que o *Wassail* possui 64% de precisão em suas avaliações de casos de teste, a abordagem apresentada alcança precisões de até 99% para a classificação, demostrando que a abordagem de calculo de frequências de características é mais adequada e representativa para a avaliação dos módulos WebAssembly em relação à abordagem de análise por meio de grafos de fluxo de controle.

Em comparação com a ferramenta proposta por (Brito et al., 2022), que utiliza análise estática de fluxo de dados para identificar vulnerabilidades em módulos WebAssembly, a abordagem proposta neste estudo não apresenta dificuldades para lidar com estruturas de dados definidas pelo usuário, pelo contrário, ela utiliza essas informações para caracterizar os módulos e realizar a análise.

A vantagem da abordagem de utilização de algoritmos ML para análise estática de códigos apresenta-se principalmente na capacidade dos modelos para aprenderem padrões de algoritmos maliciosos. Por meio da aprendizagem de padrões dos códigos binários é possível garantir maior segurança para os usuários pois o sistema de detecção não dependerá de assinaturas de algoritmos conhecidos para realizar a detecção, mas poderá também classificar e proteger

contra novos binários maliciosos, que ainda não possuem uma assinatura conhecida mas que seguem os mesmo padrões de comportamento, com alta taxa de confiabilidade.

Outra vantagem presente nesta abordagem é a extração de características de códigos já compilados, sem a dependência de acesso ao código fonte para a realização da classificação. A capacidade de utilização de binários para a classificação faz com que os modelos classificadores não sejam restritos à avaliação de códigos escritos em apenas uma linguagem, dependendo apenas da existência da seção DWARF, criada no momento de compilação, para posterior extração das características. Além disso, por tratar-se de um formato de dados padronizado para representar informações de depuração, a mesma estratégia de extração utilizada na abordagem com módulos que foram compilados a partir de código escrito em C e C++ pode ser utilizada para módulos escritos em Ruby e outras linguagens.

No entanto, é importante ressaltar algumas limitações da solução proposta. É necessário considerar que os modelos de aprendizado de máquina dependem de conjuntos de treinamento representativos e atualizados para manter sua eficácia. A constante evolução e sofisticação das técnicas de ataques cibernéticos podem resultar em variantes de código malicioso que não foram incluídas no conjunto de treinamento, levando a falsos negativos e à capacidade limitada do modelo em detectar essas novas ameaças.

Além disso, a abordagem proposta é baseada na análise estática dos binários WebAssembly, o que significa que não leva em consideração o contexto de execução dinâmica. Embora a análise estática seja eficaz na detecção de padrões e características de código malicioso, ela pode falhar em identificar ameaças que só se manifestam durante a execução, como comportamentos suspeitos em tempo real ou exploração de vulnerabilidades específicas do ambiente de execução.

A aplicação de técnicas de aprendizado de máquina para a detecção de anomalias em binários WebAssembly mostra-se altamente eficaz e promissora. A abordagem proposta nesta pesquisa supera as limitações de outras ferramentas existentes, como a falta de capacidade de lidar com estruturas de dados complexas, dependência de ferramentas externas e foco limitado em padrões pré-definidos de vulnerabilidades. Os resultados obtidos demonstraram desempenhos de até 99% de F1Score e acurácia acima de 96%, validando a eficácia da solução proposta. Além disso, a abordagem de aprendizado de máquina permite uma detecção abrangente e flexível, não dependendo apenas de assinaturas conhecidas de algoritmos maliciosos. A extração de características de binários compilados, sem a necessidade de acesso ao código fonte, e a utilização do formato padronizado DWARF para a análise permitem a aplicação da solução em uma variedade de linguagens de programação. Em suma, a aplicação de modelos de aprendizado de máquina em binários WebAssembly para a detecção de código malicioso é uma área promissora e com grande potencial para proteger sistemas contra intrusões e ameaças cibernéticas.

6 CONCLUSÃO

Neste estudo foi abordada a segurança de aplicações WebAssembly, um formato de instruções binárias que tem ganhado destaque devido à necessidade de processamentos mais rápidos e eficientes em aplicações Web. Embora o WebAssembly ofereça vantagens em termos de desempenho e portabilidade, também apresenta vulnerabilidades que precisam ser mitigadas. Problemas como *buffer overflow* e gerenciamento inadequado de memória são desafios enfrentados pelos desenvolvedores de aplicações WebAssembly. Além disso, o surgimento de malwares que realizam *criptojacking* através de navegadores representa uma ameaça significativa, bem como a intrusão em sistemas por meio de códigos maliciosos.

Neste trabalho, foi proposta uma estratégia estática para a caracterização de aplicações WebAssembly, utilizando informações encontradas nos binários para a classificação e detecção de intrusões. A abordagem baseou-se na utilização de datasets de arquivos com amostras benignas e malignas, para realizar então a extração de características dos binários WebAssembly por meio do formato de depuração DWARF e na utilização de algoritmos de ML para análise e classificação. Os resultados obtidos foram promissores, com desempenhos de até 99% de F1Score para o modelo *RandomForest* e *Accuracy* acima de 96% para os algoritmos *AdaBoost*, *K-Nearest Neighbors*, *RandomForest* e *XGBoost* para a detecção de código malicioso.

Os resultados obtidos indicam que a área de análise estática para detecção de códigos maliciosos em módulos WebAssembly por meio da aplicação de modelos de ML é um campo promissor que ainda precisa ser explorado. A expansão do Dataset para contemplar módulos compilados a partir de códigos Ruby é uma possível direção para trabalhos futuros, bem como a investigação de novas técnicas de extração de dados para algoritmos que não possuem a seção DWARF. A continuidade da pesquisa e desenvolvimento de soluções para a segurança em aplicações WebAssembly é crucial para garantir a proteção dos usuários e a integridade dos sistemas web.

REFERÊNCIAS

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K. e Siemens, C. (2014). Drebin: Effective and explainable detection of android malware in your pocket. Em *Network and Distributed System Security*, volume 14, páginas 23–26.

Battaglineg, R. (2021). The art of WebAssembly: Build secure, portable, high-performance applications. No Starch Press.

Bosamiya, J., Lim, W. S. e Parno, B. (2022). Provably-Safe multilingual software sandboxing using WebAssembly. páginas 1975–1992.

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

Brown, D. J., Suckow, B. e Wang, T. (2002). A survey of intrusion detection systems. *Department of Computer Science, University of California, San Diego*.

Ceschin, F., Gomes, H. M., Botacin, M., Bifet, A., Pfahringer, B., Oliveira, L. S. e Grégio, A. (2020). Machine learning (in) security: A stream of problems. *arXiv preprint arXiv:2010.16045*.

Chandola, V., Banerjee, A. e Kumar, V. (2009). Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):1–58.

Delendik, Y. (2020). Dwarf for WebAssembly. `https://yurydelendik.github.io/webassembly-dwarf/`.

Disselkoen, C., Renner, J., Watt, C., Garfinkel, T., Levy, A. e Stefan, D. (2019). Position paper: Progressive memory safety for WebAssembly. Em *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, páginas 1–8.

Falliere, N. (2018). Reverse engineering WebAssembly. `https://www.pnfsoftware.com/reversing-wasm.pdf`.

Fioraldi, A., Maier, D., Eißfeldt, H. e Heuse, M. (2020). AFL++ : Combining incremental steps of fuzzing research. Em *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

Fu, W., Lin, R. e Inge, D. (2018). Taintassembly: Taint-based information flow control tracking for WebAssembly. *arXiv preprint arXiv:1802.01050*.

Khan, S. S. e Madden, M. G. (2014). One-class classification: taxonomy of study and review of techniques. *The Knowledge Engineering Review*, 29(3):345–374.

Kim, M., Jang, H. e Shin, Y. (2022). Avengers, assemble! survey of WebAssembly security solutions. Em *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, páginas 543–553. IEEE.

Kirchmayr, W., Moser, M., Nocke, L., Pichler, J. e Tober, R. (2016). Integration of static and dynamic code analysis for understanding legacy source code. Em *2016 IEEE international conference on software maintenance and evolution (ICSME)*, páginas 543–552. IEEE.

Kolosick, M., Narayan, S., Johnson, E., Watt, C., LeMay, M., Garg, D., Jhala, R. e Stefan, D. (2022). Isolation without taxation: near-zero-cost transitions for WebAssembly and SFI. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–30.

Kumar, S. (2007). Survey of current network intrusion detection techniques. *Washington Univ. in St. Louis*, páginas 1–18.

Lattner, C. e Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis and transformation. páginas 75–88, San Jose, CA, USA.

Lehmann, D., Kinder, J. e Pradel, M. (2020). Everything old is new again: Binary security of WebAssembly. Em *29th USENIX Security Symposium (USENIX Security 20)*, páginas 217–234.

Lehmann, D. e Pradel, M. (2019). Wasabi: A framework for dynamically analyzing WebAssembly. Em *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, páginas 1045–1058.

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

Lehmann, D., Torp, M. T. e Pradel, M. (2021). Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of WebAssembly. *arXiv preprint arXiv:2110.15433*.

Liu, M., Xue, Z., Xu, X., Zhong, C. e Chen, J. (2018). Host-based intrusion detection system with system calls: Review and future trends. *ACM Computing Surveys (CSUR)*, 51(5):1–36.

Michael, A. E., Gollamudi, A., Bosamiya, J., Disselkoen, C., Denlinger, A., Watt, C., Parno, B., Patrignani, M., Vassena, M. e Stefan, D. (2022). Mswasm: Soundly enforcing memory-safe execution of unsafe code. *arXiv preprint arXiv:2208.13583*.

Naseem, F. N., Aris, A., Babun, L., Tekiner, E. e Uluagac, A. S. (2021). Minos: A lightweight real-time cryptojacking detection system. Em *Network and Distributed System Security*.

Pistoia, M., Chandra, S., Fink, S. J. e Yahav, E. (2007). A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM systems journal*, 46(2):265–288.

Quan, L., Wu, L. e Wang, H. (2019). Evulhunter: Detecting fake transfer vulnerabilities for EOSIO's smart contracts at WebAssembly-level.(2019). *arXiv preprint arXiv:1906.10362*.

Romano, A., Lehmann, D., Pradel, M. e Wang, W. (2022). Wobfuscator: Obfuscating javascript malware via opportunistic translation to WebAssembly. Em *Proceedings of the 2022 IEEE Symposium on Security and Privacy (S&P 2022)*, páginas 1101–1116.

Romano, A. e Wang, W. (2020). Wasim: Understanding WebAssembly applications through classification. Em *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, páginas 1321–1325. IEEE.

Rossberg, A. (2018). WebAssembly specification. `https://webassembly.github.io/spec/core/_download/WebAssembly.pdf`.

Shaukat, K., Luo, S., Varadharajan, V., Hameed, I. A. e Xu, M. (2020). A survey on machine learning techniques for cyber security in the last decade. *IEEE Access*, 8:222310–222354.

Stiévenart, Q. e De Roover, C. (2020). Compositional information flow analysis for WebAssembly programs. Em *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, páginas 13–24. IEEE.

Stiévenart, Q., De Roover, C. e Ghafari, M. (2022). Security risks of porting C programs to WebAssembly. Em *Proceedings of the 37th Symposium on Applied Computing*, páginas 1713–1722, Virtual Event. ACM.

Szanto, A., Tamm, T. e Pagnoni, A. (2018). Taint tracking for WebAssembly. *arXiv preprint arXiv:1807.08349*.

Watt, C., Renner, J., Popescu, N., Cauligi, S. e Stefan, D. (2019). Ct-wasm: type-driven secure cryptography for the web ecosystem. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29.

Zhang, J., Roesner, F. e Wagner, D. (2020). Autonomous systems security: Static analysis for embedded systems. *IEEE Security & Privacy*, 18(1):12–22.