Uso de Chamadas WASI para a Identificação de Ameaças em Aplicações WebAssembly

Tiago Heinrich[1], **Newton C. Will**[2], **Rafael R. Obelheiro**[3], **Carlos A. Maziero**[1]

[1] Departamento de Informática
Universidade Federal do Paraná (UFPR)
Curitiba – PR – Brasil

[2]Departamento de Ciência da Computação
Universidade Tecnológica Federal do Paraná (UTFPR)
Dois Vizinhos – PR – Brasil

[3]Programa de Pós-Graduação em Computação Aplicada (PPGCAP)
Universidade do Estado de Santa Catarina (UDESC)
Centro de Ciências Tecnológicas – Joinville – SC – Brasil

{theinrich,maziero}@inf.ufpr.br, will@utfpr.edu.br,
rafael.obelheiro@udesc.br

Abstract. *WebAssembly (or Wasm) is a bytecode format that has gained fast adoption due to good performance, compact representation, and portability. It is mostly used as a compilation target for high-level programming languages such as C, C++, Go, and Rust, and may be executed within web browsers or native runtimes. Although security is one of the design goals for WebAssembly, there remain issues with malicious code, especially for web applications. In this paper, we introduce a method for performing anomaly-based detection of malicious Wasm binaries through dynamic analysis. We propose a classification of WASI calls – the Wasm counterpart of system calls – according to their risk and function and use it to categorize the calls issued by Wasm binaries, which allows us to detect malicious binaries using machine learning models. Our results show that this is a promising approach for identifying malicious WebAssembly code.*

Resumo. *WebAssembly (ou Wasm) é um formato de* bytecode *que vem ganhando rápida adoção devido a seu bom desempenho, representação compacta, e portabilidade. Ele é mais usado como alvo de compilação para linguagens de programação de alto nível, como C, C++, Go e Rust, podendo ser executado dentro de navegadores Web ou em* runtimes *nativos. Embora a segurança seja uma meta do projeto do WebAssembly, ainda existem problemas com código malicioso, especialmente para aplicações Web. Este artigo introduz um método para realizar detecção baseada em anomalias de binários Wasm maliciosos, por meio de análise dinâmica. É proposta uma classificação de chamadas WASI – equivalentes no Wasm a chamadas de sistema – de acordo com seu risco e funcionalidade, a qual é usada para categorizar as chamadas realizadas por binários Wasm, o que permite detectar binários maliciosos usando modelos de aprendizagem de máquina. Os resultados obtidos mostram que esta é uma abordagem promissora para a identificação de código WebAssembly malicioso.*

1. Introdução

WebAssembly é um *bytecode* de baixo nível para execução em máquinas virtuais, que oferece portabilidade, representação compacta e bom desempenho [Battagline 2021]. Várias linguagens de programação (incluindo C, C++, Rust e Go) podem ser usadas para gerar código WebAssembly, o qual pode ser executado dentro de navegadores Web (da mesma forma que JavaScript) ou em ambientes de execução nativos. O WebAssembly é suportado pelos principais navegadores, incluindo Chrome, Edge, Firefox e Safari.

Em alguns cenários, como em aplicações Web, o uso de WebAssembly traz código de procedência incerta para ser executado na máquina de um cliente, o que exige um ambiente seguro de execução. Tendo isso em vista, a execução de código WebAssembly ocorre em isolamento, dentro de uma *sandbox*. Mesmo assim, foram identificadas algumas preocupações relacionadas à segurança de aplicações WebAssembly, incluindo problemas referentes à segurança de memória por conta do suporte a linguagens como C e C++ e com código malicioso ofuscado e/ou que realiza mineração de criptomoedas (*cryptojacking*) [Kim et al. 2022].

A mitigação dos riscos associados a código Wasm pode envolver mecanismos de prevenção (como *hardening* do ambiente de execução) e detecção de código malicioso. A detecção de código malicioso pode usar análise estática, identificando vulnerabilidades e instruções indevidas em código fonte ou binário, e/ou análise dinâmica, monitorando o comportamento do código durante sua execução e identificando ações indevidas ou anômalas [Aggarwal and Jalote 2006, Lemos et al. 2022]. Uma classe importante de comportamento malicioso envolve o acesso a recursos do sistema cliente, como arquivos, dispositivos de entrada e saída e outros processos, para comprometer a confidencialidade, integridade ou disponibilidade de dados ou do próprio sistema. Isso requer o uso de chamadas WebAssembly System Interface (WASI), que estabelecem como código Wasm interage com o sistema operacional [Battagline 2021].

Este trabalho propõe o uso de análise dinâmica de código para a identificação de ameaças em aplicações WebAssembly. A solução examina as chamadas WASI efetuadas por código WebAssembly, as quais são categorizadas para fins de aprendizado e classificação com o uso de modelos de aprendizado de máquina. As contribuições deste trabalho são:

- Uma investigação do uso de chamadas WASI para a identificação de ameaças em WebAssembly;
- Uma classificação categórica que pode ser utilizada tanto para chamadas WASI quanto para chamadas de sistema; e
- A proposição de modelos de aprendizado de máquina com base em dados categóricos sobre as chamadas WASI efetuadas por aplicações para a identificação de ameaças.

Este trabalho está estruturado em seis seções. A Seção 2 apresenta a fundamentação teórica necessária para o entendimento do trabalho. A Seção 3 discute trabalhos relacionados. A Seção 4 introduz a proposta do trabalho. A Seção 5 apresenta os experimentos realizados para avaliar a proposta e os resultados obtidos. Por fim, a Seção 6 conclui o trabalho.

2. Fundamentação Teórica

Esta seção revisa alguns conceitos necessários para a compreensão da proposta, abrangendo WebAssembly (Seção 2.1), análise dinâmica de aplicações (Seção 2.2) e dados categóricos (Seção 2.3).

2.1. WebAssembly

WebAssembly (também conhecido como Wasm) é um formato de código binário portável, projetado para execução segura e eficiente, com uma representação compacta [Rossberg 2022]. Ele é tipicamente usado como formato de código executável gerado por compiladores de linguagens de alto nível, como C, C++, Go e Rust, embora tenha também uma representação textual chamada WebAssembly Text (WAT). Desta forma, o WebAssembly permite que aplicações e bibliotecas desenvolvidas nessas linguagens sejam executadas em navegadores Web (alvo inicial do Wasm) ou em ambientes nativos [Hoffman 2019].

O *bytecode* WebAssembly tipicamente é executado dentro de uma *sandbox* chamada de WebAssembly *runtime*. Esta escolha de projeto permite com que as aplicações sejam executadas em uma gama maior de plataformas, reduz o tamanho das mensagens a serem trocadas com o servidor para recuperar conteúdo, e oferece mais segurança para o ambiente do cliente que executa o conteúdo.

Ao considerar a adoção do WebAssembly, diferentes áreas se beneficiam dos recursos oferecidos pelo *design* do formato, alguns deles sendo: criptografia, vídeo, áudio, e gráficos. O uso do WebAssembly não está limitado ao ambiente Web, sendo aplicado tanto para o lado do servidor, serviço de distribuição e aplicativos móveis [Rossberg 2022].

A WebAssembly System Interface (WASI) é uma Application Programming Interface (API) que define as regras para o WebAssembly *runtime* e interações que serão realizadas com o sistema operacional (SO) pela aplicação. Também permite a execução de operações nativas sem a quebra nos níveis de segurança já definidos para aplicações WebAssembly [Battagline 2021].

As chamadas WASI são responsáveis pela interação entre a aplicação WebAssembly e o SO, possuindo um comportamento similar ao de chamadas de sistema, que são utilizadas por aplicações nativas para invocar serviços do SO. Chamadas WASI permitem que aplicações acessem recursos tais como sistema de arquivos, rede e entrada/saída (E/S) com o usuário. Atualmente 45 chamadas WASI são suportadas, com previsão de que mais chamadas sejam adicionadas em futuras versões [BytecodeAlliance 2021].

A Figura 1 ilustra a relação entre chamadas WASI e chamadas de sistema. Quando uma aplicação nativa (app.exe) precisa acessar recursos do SO, ela invoca uma função da biblioteca C nativa (glibc), que por sua vez invoca as chamadas de sistema oferecidas pelo SO. Uma aplicação WebAssembly (app.wasm) executando em um *runtime* nativo invoca uma função da WASI libc, que usa a WASI API para acessar a glibc e daí as chamadas de sistema. Quando o código Wasm é executado em um navegador Web, isso ocorre dentro de um Browser Engine, o que faz com que chamadas WASI estejam sujeitas a passar por mais camadas de código até que cheguem às chamadas de sistema.

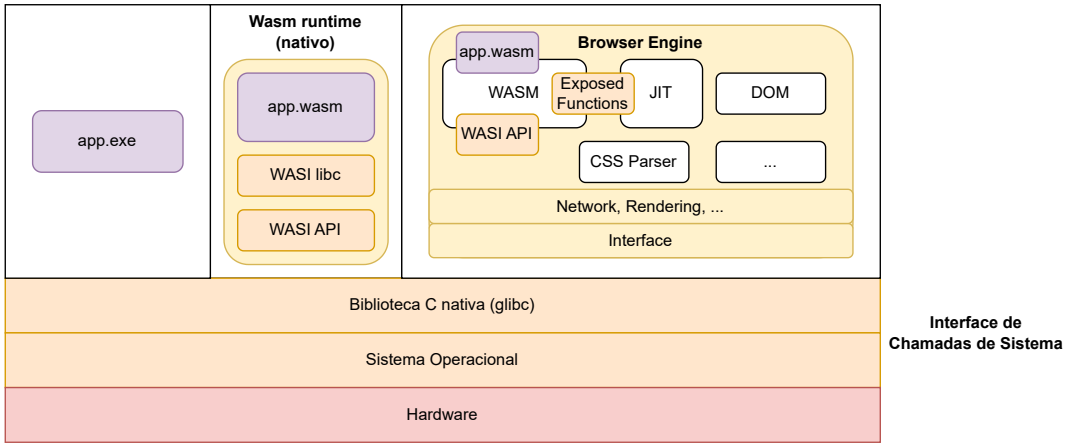

Figura 1. Relação entre chamadas de sistema e chamadas WASI.

2.2. Análise Dinâmica de Aplicações WebAssembly

Uma das formas de identificar processos maliciosos em um sistema é analisando o seu comportamento durante a execução, ou análise dinâmica [Aggarwal and Jalote 2006]. Em comparação com a análise estática, que busca identificar comportamento malicioso analisando as instruções em código fonte ou binário, a análise dinâmica traz alguns benefícios, como ser pouco impactada por ofuscação de código e permitir a observação de comportamentos emergentes que não são diretamente visíveis no código (porque dependem de dados de entrada, por exemplo).

Um método de análise dinâmica consiste em analisar as chamadas de sistema [Forrest et al. 1996, Castanhel et al. 2021, Lemos et al. 2023] ou de comunicação inter-processos [Lemos et al. 2023] realizadas por um processo. A premissa é que existe maior risco associado a interações com componentes externos ao código, envolvendo outros processos ou recursos do sistema como arquivos e dispositivos de E/S, do que quando o processo fica inteiramente contido em seu espaço de endereçamento.

No caso de código WebAssembly, interações com o ambiente externo à *sandbox* são realizadas por meio de chamadas WASI, que acabam sendo traduzidas em chamadas de sistema. Assim, o comportamento do código pode ser analisado tanto considerando suas chamadas WASI quanto suas chamadas de sistema. No entanto, não existe um mapeamento 1:1 entre chamadas WASI e chamadas de sistema. Além disso, durante a execução de código Wasm, nem todas as chamadas de sistema emitidas pelo processo que encapsula a *sandbox* referem-se necessariamente a tal código, uma vez que o *runtime* também pode realizar chamadas para atender a suas próprias necessidades. Portanto, para o monitoramento de código WebAssembly, a observação de chamadas WASI pode ser mais vantajosa do que a de chamadas de sistema, uma vez que chamadas WASI estão mais perto da semântica do código e exibem menos ruído (chamadas que não provêm diretamente da aplicação) comparadas às chamadas de sistema.

A análise de comportamento pode avaliar chamadas isoladas ou sequências de chamadas, tipicamente considerando uma janela temporal. Os parâmetros das chamadas também podem ou não ser usados na avaliação. A escolha da abordagem adotada envolve alguns *trade-offs*. Por exemplo, sequências de chamadas tendem a caracterizar melhor os comportamentos do que chamadas isoladas [Forrest et al. 1996], mas impõem um maior

overhead de armazenamento e processamento. Da mesma forma, usar os parâmetros de chamadas na análise oferece mais contexto e um maior detalhamento, o que tem tanto aspectos positivos (melhor delimitação de comportamentos) quanto negativos (maior *overhead* de armazenamento e processamento, dados mais ruidosos no caso de parâmetros que são diferentes a cada execução). No caso de detecção baseada em anomalias, é comum o uso de algoritmos de aprendizado de máquina para classificação dos dados. A abordagem tradicional emprega aprendizado supervisionado, usando conjuntos de dados rotulados (códigos benignos e maliciosos) para treinamento e avaliação dos classificadores.

2.3. Dados Categóricos

Dados categóricos, ou variáveis categóricas, são definidas como variáveis que são medidas pela categorização de um conjunto limitado de "valores" [Powers and Xie 2008]. Abordagens categóricas são comuns no campo das ciências sociais; algumas das informações representadas por categorias são gênero, idade, idioma e localização.

Há uma variedade de tipos de medição [Powers and Xie 2008], conforme apresentado na Figura 2. Uma variável categórica assume um rótulo em um conjunto discreto de valores existentes. Uma variável pode ser qualitativa ou quantitativa, e diferentes representações podem ser utilizadas para a categorização. Por exemplo, um endereço Internet Protocol (IP) pode ser categorizado como *privado* ou *público*, onde um endereço privado é um endereço que deve ser usado apenas em redes internas e um endereço público é roteável globalmente. Este é um exemplo de uma categorização nominal qualitativa. Os endereços IP também podem ser categorizados de acordo com seu endereço de rede: 192.0.2.4 e 192.0.2.51 pertencem à categoria 192.0.2.0/24, enquanto 128.9.2.51 pertence à categoria 128.9.0.0/16. Este é um exemplo de categorização ordinal qualitativa (endereços de rede podem ser ordenados e é possível definir uma medida de distância entre redes). O uso de categorias para descrever endereços IP pode fornecer uma representação mais compacta dos dados (1 bit é suficiente para diferenciar entre *público* e *privado*), pode levar a regras de decisão mais simples (por exemplo, em uma política de controle de acesso à rede, pode-se escrever regras que se aplicam a todos os endereços IP *privados* ou *públicos*), ou ainda complementar ou substituir os dados brutos em tarefas de classificação de dados; esta última forma é a proposta deste trabalho.

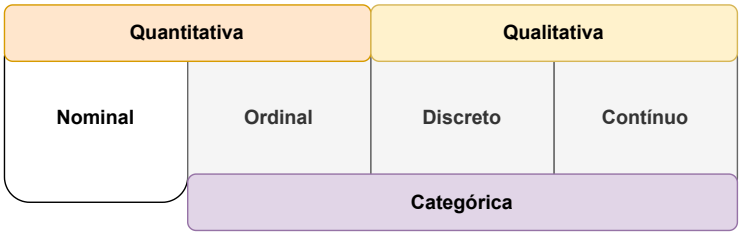

Figura 2. Tipos de medidas e relação com dados categóricos. Baseado em [Powers and Xie 2008].

3. Trabalhos Relacionados

Esta seção discute trabalhos relacionados envolvendo análise estática e dinâmica de código WebAssembly.

Wasabi é um *framework* para análise dinâmica de binários WebAssembly [Lehmann and Pradel 2019]. A ferramenta permite a análise de instruções, grafo de chamadas, além de recursos para análise como *taint* e *tracing*. Uma API oferecida pelo *framework* permite que novos sistemas de avaliação sejam implementados e consigam extrair informações para posteriores processos de detecção e análise dos binários. Um fator negativo da proposta consiste no *overhead* observado, que pode alcançar um aumento de até 163 vezes no tempo de execução.

WASim é uma ferramenta de análise estática que classifica binários WebAssembly de acordo com seu propósito [Romano and Wang 2020]. Cada binário é classificado em uma de onze classes (como jogo, aplicação gráfica, ou ferramenta de compressão) a partir de características do *bytecode* e da representação textual (WAT) correspondente. As características incluem nomes de funções importadas e exportadas, e vários atributos de tamanho. Os binários WebAssembly usados para a avaliação foram coletados de sítios Web no ranking Top 1M da Alexa (o Top 1M representa o milhão de sítios Web mais populares). Foi obtido um *dataset* com 734 amostras únicas, que foi utilizado para o treinamento e avaliação do modelo proposto. Quatro algoritmos de aprendizado de máquina foram avaliados (Naïve Bayes, Random Forest, Support Vector Machine e rede neural), com os melhores valores para o *f1score* sendo 91,6% para rede neural e 87,8% para o SVM.

Wasmati é uma ferramenta para a identificação de vulnerabilidades em binários WebAssembly por meio de análise estática [Brito et al. 2022]. Para cada binário é gerado um *code property graph* (CPG), o que permite capturar estruturas, dados e fluxos de controle, apresentando relevância para um processo de identificação de vulnerabilidades. Para a investigação de vulnerabilidades, dez consultas foram definidas, representando as vulnerabilidades que o Wasmati busca nos binários. A ferramenta identificou 83 k vulnerabilidades em 3,1 k binários WebAssembly únicos.

[Helpa et al. 2023] apresenta uma estratégia estática para a detecção de anomalias em binários WebAssembly. A estratégia utiliza o *Debugging With Attributed Record Formats* (DWARF) para extrair informações de binários WebAssembly que posteriormente foram utilizados para treinar e testar modelos de aprendizado de máquina para a detecção de anomalias [Delendik 2020]. Os resultados alcançados destacam um campo promissor para a detecção de ameaças através da extração de informações dos binários.

TaintAssembly é um mecanismo voltado ao rastreamento de funções e da memória linear de código WebAssembly [Fu et al. 2018]. A estratégia aplica o rastreamento através do *taint*, onde dados podem ser rastreados durante o processo de execução da aplicação (análise dinâmica). Esta proposta apresenta um *overhead* de até 12%, e é favorável para depuração de aplicações.

MineThrottle é um mecanismo de defesa para detectar e bloquear aplicações WebAssembly que exploram *cryptojacking* [Bian et al. 2020]. A proposta foi avaliada com páginas no ranking Top 1M da Alexa, alcançando uma taxa de 1,8% de falsos positivos. Ao todo, foram observadas 109 páginas Web no Top 1M com comportamento de *cryptojacking*.

As experiências na literatura voltadas à detecção de ameaças em aplicações *WebAssembly* são ainda limitadas. De modo geral, elas focam em tipos específicos de ataques, como *cryptojacking*, ou na identificação de erros no desenvolvimento das aplicações. Não foram encontradas referências que visem à detecção de anomalias com base em chamadas

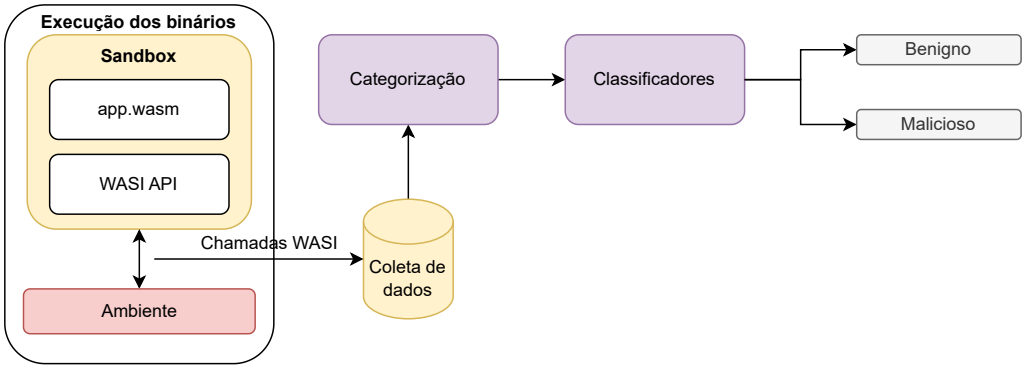

Figura 3. Visão geral da proposta de detecção de intrusões usando chamadas WASI

WASI ou chamadas de sistema, como explorado neste trabalho.

4. Proposta

Esta seção apresenta nossa proposta para detecção de anomalias em aplicações WebAssembly usando dados categóricos com base nas chamadas WASI emitidas pelo código. A Seção 4.1 dá uma visão geral da proposta, e a Seção 4.2 detalha como as chamadas são categorizadas.

4.1. Visão geral

A Figura 3 ilustra a visão geral da proposta. O objetivo é realizar detecção *off-line* de código WebAssembly malicioso. O processo consiste em executar o código e registrar as chamadas WASI realizadas durante a execução. Após a execução, as chamadas são categorizadas de acordo com seu nível de risco e funcionalidade, segundo o esquema apresentado na Seção 4.2. As categorias associadas às chamadas são então repassadas a um classificador de aprendizado de máquina previamente treinado, que irá indicar se o binário é benigno ou malicioso. A figura omite a fase de treinamento dos classificadores; a Seção 5 discute como foram treinados os classificadores nos nossos experimentos.

4.2. Categorização de Chamadas

A categorização de dados permite que características de operações e funcionalidades sejam associadas a grupos similares. Estas categorizações posteriormente podem ser utilizadas no processo de aprendizado, para destacar características que não estariam presentes nos dados brutos. Esta estratégia contribui para modelos de classificação onde características categóricas podem ser utilizadas no processo de aprendizado.

A classificação proposta visa ser aplicada na classificação de chamadas WASI. Todavia, dada a proximidade semântica entre chamadas WASI e chamadas de sistema, uma classificação que funcione para ambos os tipos é interessante, pois facilita uma eventual comparação entre eles. Duas classificações são propostas aqui, uma considerando o risco associado a uma chamada e outra considerando as funcionalidades implementadas pelas chamadas.

A Tabela 1 apresenta a classificação de acordo com o risco oferecido pelas chamadas WASI. Os quatro níveis foram definidos em função da observação de chamadas WASI no conjunto de amostras de código Wasm empregado no experimento (Seção 5.1). O nível

representa correlação, não causalidade. Portanto, não é porque um código possui uma chamada classificada em um dos níveis considerados de alto risco (A, B ou C) que ele é malicioso/vulnerável. Por outro lado, uma aplicação que usa várias chamadas na classe A (por exemplo) tende a oferecer mais riscos do que uma aplicação que usa apenas chamadas na classe D.

Tabela 1. Classificação de acordo com o risco das chamadas.

Risco	Nível	Tipo de operação
Alto	A	chamadas que, sozinhas, aparecem com mais frequência em código malicioso/vulnerável do que em código benigno
	B	chamadas que foram usadas de forma conjunta com mais frequência em código malicioso/vulnerável do que em código benigno
	C	chamadas que, em código malicioso/vulnerável, são repetidas muito mais vezes do que em código benigno
Baixo	D	chamadas que são usadas da mesma forma em código malicioso/vulnerável e em código benigno, e cuja semântica permite supor que não possam provocar uma violação de segurança

A Tabela 2 traz uma classificação complementar, baseada na função que cada chamada executa. Ela é adaptada de uma classificação de chamadas de sistema para o Linux 2.2 apresentada em [Bernaschi et al. 2002]. Foram realizadas duas adaptações nessa classificação. As chamadas de manipulação de arquivos e de dispositivos, que estavam no mesmo grupo, foram separadas em dois grupos, um com as chamadas referentes a arquivos (grupo 1) e outro com as chamadas referentes a dispositivos (grupo 10). A separação se justifica porque os SOs atuais possuem um conjunto mais rico de chamadas para manipulação de dispositivos, e a distinção entre arquivos e dispositivos é mais acentuada nas chamadas WASI. Outra alteração foi no grupo 9, que continha apenas chamadas não implementadas e agora inclui também chamadas removidas ou usadas para depuração do *kernel*.

Tabela 2. Classificação de Funcionalidades.

Grupo	Funcionalidades
1	Manipulação de arquivo
2	Controle de processos
3	Gerenciamento de módulos
4	Gerenciamento de memória
5	Operação de tempo
6	Comunicação
7	Informações do sistema
8	Reservado
9	Não implementado/removido ou depuração
10	Manipulação do dispositivo

As classificações das Tabelas 1 e 2 são usadas em conjunto no processo de detecção de intrusões. As chamadas WASI emitidas pela aplicação são mapeadas em uma categoria

formada por seu nível de risco e funcionalidade. Assim, uma chamada de controle de processo que sozinha oferece alto risco será representada pela categoria A-2. Esta categorização, que destaca as chamadas que representam maior risco, é usada nas etapas de treinamento e avaliação dos modelos de aprendizado de máquina.

A Tabela 3 mostra a categorização das chamadas WASI usando as duas classificações de forma combinada. Observa-se que os grupos 3, 4 e 8 (definidos na Tabela 2) estão vazios. Optou-se por manter esses grupos tanto para facilitar a correspondência com a classificação de [Bernaschi et al. 2002] quanto contemplando a perspectiva de introdução de novas chamadas WASI: pela sua natureza, é provável que eventuais novas chamadas ofereçam funcionalidades compatíveis com as de chamadas de sistema existentes.

Tabela 3. Classificação de chamadas WASI.

Nível	Gr	Chamadas WASI
A	1	`path_link`, `path_rename`, `path_symlink`, `path_unlink_file`, `path_remove_directory`, `path_filestat_set_times`, `args_get`, `environ_get`
B	1	`fd_fdstat_set_flags`, `fd_tell`, `fd_seek`, `path_create_directory`, `fd_pread`, `fd_pwrite`, `sock_recv`, `fd_read`, `sock_send`, `fd_write`, `fd_filestat_get`, `path_create_directory`
	2	`fd_advise`, `sched_yield`
C	1	`fd_renumber`, `fd_allocate`, `path_open`, `random_get`
	6	`sock_recv`, `sock_send`, `proc_raise`
D	1	`fd_close`, `path_readlink`, `path_filestat_get`, `args_sizes_get`, `environ_sizes_get`, `fd_prestat_get`, `fd_prestat_dir_name` `fd_readdir`
	5	`clock_res_get`, `clock_time_get`
	7	`sock_shutdown`
	9	`fd_filestat_set_times`
	10	`fd_datasync`, `fd_sync`, `fd_fdstat_get`

A aplicação deste conjunto de categorias oferece as seguintes vantagens:

- Simplificação no volume de dados utilizados e sistemas de detecção de intrusão;
- Melhor compreensão dos comportamentos apresentados em cada aplicação; e
- Um modelo que é flexível tanto para aplicações WebAssembly pelo uso de chamadas WASI, como para aplicações nativas pelo uso de chamadas de sistema.

5. Avaliação

Esta seção apresenta a avaliação da proposta. A Seção 5.1 descreve os objetivos da avaliação e os experimentos realizados. A Seção 5.2 analisa os resultados obtidos, e a Seção 5.3 faz uma discussão dos resultados.

5.1. Objetivos e experimentos realizados

Os experimentos têm por objetivo avaliar a eficácia do uso de dados categóricos sobre chamadas WASI na detecção de código WebAssembly malicioso. A detecção utiliza

modelos de aprendizado de máquina para o processo de classificação. Cinco classificadores foram escolhidos para a avaliação: XGBoost, NuSVC, RadiusNeighbors, AdaBoost e *stochastic gradient descent* (SGD). Os algoritmos selecionados são populares para detecção de intrusões baseada em anomalias, e sua diversificação permite que se observe como a proposta se comporta com estratégias variadas de classificação [Mishra et al. 2018, Ceschin et al. 2020].

Os experimentos requerem um conjunto de dados representativo de aplicações WebAssembly. A disponibilidade de dados que possam ser utilizados para estudos com WebAssembly é pequena [Hilbig et al. 2021, Stiévenart et al. 2021, Stiévenart et al. 2022]. A disponibilidade de dados que possam ser executados para análise dinâmica é ainda menor, já que na literatura existe um foco na análise estática de binários Wasm. Assim, foi necessário criar um *dataset* contendo amostras compatíveis com diversos compiladores WebAssembly.

O *dataset* utilizado foi coligido de diversas fontes. O primeiro conjunto de dados utilizado foi [Stiévenart et al. 2022], que consiste de binários Wasm que possuem problemas de implementação e segurança que persistiram do porte de aplicações C para o WebAssembly. O segundo conjunto de dados utilizado, que complementa o primeiro conjunto, consiste de uma variedade de *benchmarks*, *test suites* e aplicações [Stiévenart 2021, Denis 2023, WebAssembly 2023, Beyer 2023], todos de fontes públicas e utilizados pela comunidade. Os binários obtidos foram manualmente selecionados para eliminar códigos duplicados e muito semelhantes, com o intuito de preservar a diversidade. Desta forma, o dataset contem uma variedade de padrões de ataques, como comportamentos normais esperado por aplicações WebAssembly. O *dataset* final possui 262 amostras benignas e 261 amostras maliciosas[1].

Para a realização dos experimentos, os binários que formam o dataset foram executados em um ambiente sandbox WebAssembly, especificamente foi utilizado o Wasmtime v6.0.0. O conjunto de dados gerados contem o trace da execução das aplicações, com as respectivas chamadas WASI. Posteriormente as chamadas WASI foram categorizadas de acordo com a estratégia apresentada na Seção 4.2. O conjunto de dados foi dividido ao meio, com a primeira porção sendo utilizada para o treinamento dos modelos e a segunda parte para o teste dos algoritmos. Os parâmetros padrões de cada modelo foram utilizados para o treinamento. Os modelos foram treinados com os dados categorizados, utilizando as classificações de operação e funcionalidade apresentadas na Seção 4.2.

5.2. Análise de dados

A Tabela 4 apresenta os resultados obtidos pelos cinco classificadores avaliados. No geral, os resultados alcançados são promissores, com todos algoritmos alcançando um *F1score* acima de 96%, destacando que os modelos conseguiram aprender as características do conjunto de dados utilizado através da estratégia de categorização de chamadas WASI.

Ao averiguar o *recall*, nota-se que o SGD é o único modelo com falsos negativos, que por consequência acaba afetando o valor alcançado para o *f1score*. A *precision* mostra a incidência de falsos positivos; nos experimentos, quatro dos cinco algoritmos tiveram baixa incidência, com *precision* igual ou superior a 97,67%, indicando que os

[1]O conjunto de dados pode ser encontrado em `https://github.com/h31nr1ch/wasm-binary-database/releases/tag/v1.0`.

Tabela 4. Resultado alcançado pelos classificadores.

Algoritmo	F1Score	Recall	Precision	Brier Score	BAC
XGBoost	98,82%	100%	97,67%	1,15%	98,90%
NuSVC	96,18%	100%	92,65%	3,82%	96,32%
RadiusNeighbors	98,82%	100%	97,67%	1,15%	98,90%
AdaBoost	98,82%	100%	97,67%	1,15%	98,90%
SGD	99,96%	99,60%	100%	3,80%	99,60%

modelos conseguiram aprender adequadamente os grupos para a classificação. A exceção foi o NuSVC, com *precision* de 92,65%. As classificações erradas que geraram os falsos positivos são responsáveis pela redução do *f1score*.

O *brier score* demonstra que os modelos estão balanceados, com apenas dois modelos encontrando valores acima de 2%. O *balanced accuracy* (BAC) reforça a adequação dos modelos, já que a métrica reflete o impacto de verdadeiros positivos e verdadeiros negativos nos classificadores.

Os resultados mostram a eficácia da estratégia de detecção proposta. Eles revelam ainda que todos os algoritmos tiveram bom desempenho, não havendo nenhum indiscutivelmente melhor do que os demais. SGD destacou-se em *f1score* e *precision*, mas foi o único que teve falsos negativos e foi um dos dois modelos menos balanceados. O NuSVC teve um desempenho pior do que o dos outros modelos em quatro das cinco métricas, podendo ser considerado o menos indicado dos cinco algoritmos avaliados.

Os resultados alcançados destacam a eficácia da categorização de chamadas proposta (Seção 4.2), também permitindo o entendimento no uso de dados categóricos para estratégias voltadas na identificação de anomalias. Com resultados promissores para a aplicação de estratégias voltadas a classificação para a identificação de ameaças e seu uso com modelos de aprendizado de máquina.

5.3. Discussão

A proposta apresentada para a detecção de anomalias traz contribuições tanto pela observação de chamadas WASI para análise dinâmica de código WebAssembly quanto pelo uso de dados categóricos no processo de classificação. As chamadas WASI permitem monitorar as interações do código Wasm com o ambiente externo à *sandbox* de execução, e a categorização dos dados torna mais simples os modelos de ML. Cabe destacar que a discussão no uso de dados categóricos para estratégias voltadas à segurança é, até o momento, limitada. A eficácia da estratégia proposta aponta que a categorização dos dados neste meio apresenta resultados promissores.

Para o processo de identificação de anomalias em WebAssembly, os resultados obtidos destacam um novo ponto de vista, que consiste em explorar as chamadas WASI para realizar a classificação de binários Wasm. Esta solução é importante devido ao uso do *sandbox* WebAssembly para a execução de aplicações em diversos ambientes, que por consequência poderiam explorar esta informação intermediária que o ambiente fornece para a identificação de ameaças.

6. Conclusão

Este trabalho apresentou uma estratégia voltada na identificação de anomalias em aplicações WebAssembly através da observação de chamadas WASI. Devido às aplicações WebAssembly serem executadas em uma *sandbox*, interações com o ambiente exterior geram chamadas WASI, que podem ser categorizadas para o processo de identificação de anomalias. Os resultados alcançados mostram que o observação de chamadas WASI e o uso de dados categóricos são promissores para a identificação de anomalias.

Para trabalhos futuros, um conjunto de dados maior será gerado para continuar explorando o uso de chamadas WASI para a identificação de anomalias em aplicações WebAssembly. Pretende-se também avaliar experimentalmente a detecção de anomalias usando chamadas de sistema com base na classificação introduzida neste artigo, bem como adaptar a proposta para a detecção *online* de anomalias em código WebAssembly.

Agradecimentos

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

Referências

Aggarwal, A. and Jalote, P. (2006). Integrating static and dynamic analysis for detecting vulnerabilities. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 1, pages 343–350. IEEE.

Battagline, R. (2021). *The Art of WebAssembly: Build Secure, Portable, High-Performance Applications*. No Starch Press, San Francisco, CA, USA.

Bernaschi, M., Gabrielli, E., and Mancini, L. V. (2002). REMUS: A security-enhanced operating system. *ACM Transactions on Information and System Security (TISSEC)*, 5(1):36–61.

Beyer, C. (2023). Amalgamated WebAssembly System Interface test suite. `https://github.com/caspervonb/wasi-test-suite`.

Bian, W., Meng, W., and Zhang, M. (2020). MineThrottle: Defending against Wasm in-browser cryptojacking. In *Proceedings of the 29th The Web Conference*, pages 3112–3118, Taipei, Taiwan. ACM.

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

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

Castanhel, G. R., Heinrich, T., Ceschin, F., and Maziero, C. (2021). Taking a peek: An evaluation of anomaly detection using system calls for containers. In *2021 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6. IEEE.

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

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

Denis, F. (2023). webassembly-benchmarks. https://github.com/jedisct1/webassembly-benchmarks.

Forrest, S., Hofmeyr, S. A., Somayaji, A., and Longstaff, T. A. (1996). A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128. IEEE.

Fu, W., Lin, R., and Inge, D. (2018). TaintAssembly: Taint-based information flow control tracking for WebAssembly.

Helpa, C., Heinrich, T., Botacin, M., Will, N. C., Obelheiro, R. R., and Maziero, C. A. (2023). Uma estratégia dinâmica para a detecção de anomalia em binários WebAssembly. In *Anais Estendidos do XXIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*. SBC.

Hilbig, A., Lehmann, D., and Pradel, M. (2021). An empirical study of real-world WebAssembly binaries: Security, languages, use cases. In *Proceedings of the 30th The Web Conference*, pages 2696–2708, Ljubljana, Slovenia. ACM.

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

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

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

Lemos, R., Heinrich, T., Maziero, C. A., and Will, N. C. (2022). Is it safe? identifying malicious apps through the use of metadata and inter-process communication. In *2022 IEEE International Systems Conference (SysCon)*, pages 1–8. IEEE.

Lemos, R., Heinrich, T., Will, N. C., Obelheiro, R. R., and Maziero, C. A. (2023). Inspecting Binder transactions to detect anomalies in Android. In *Proceedings of the 17th Annual IEEE International Systems Conference*, Vancouver, BC, Canada. IEEE.

Mishra, P., Varadharajan, V., Tupakula, U., and Pilli, E. S. (2018). A detailed investigation and analysis of using machine learning techniques for intrusion detection. *IEEE communications surveys & tutorials*, 21(1):686–728.

Powers, D. and Xie, Y. (2008). *Statistical methods for categorical data analysis*. Emerald Group Publishing.

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

Rossberg, A. (2022). WebAssembly specification. https://webassembly.github.io/spec/core/index.html.

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

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

Stiévenart, Q. (2021). SAC 2022 dataset. figshare. `https://doi.org/10.6084/m9.figshare.17297477.v1`.

WebAssembly (2023). WASI tests. `https://github.com/WebAssembly/wasi-testsuite`.