UNIVERSIDADE FEDERAL DO AMAZONAS
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Bento Pedro Chico Baptista

Identificação de Comportamento Malicioso em *WebAssembly*

Manaus
2026

Bento Pedro Chico Baptista

Identificação de Comportamento Malicioso em *WebAssembly*

Dissertação de Mestrado, apresentada ao Programa de Pós-Graduação em Informática do Instituto de Computação da Universidade Federal do Amazonas, como requisito parcial para obtenção do grau de Mestre em Informática.

Orientador: Prof. Dr. Eduardo Luzeiro Feitosa

Manaus
2026

Ficha Catalográfica

Elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

B222i Bento Pedro Chico Baptista

Identificação de Comportamento Malicioso em WebAssembly / Bento Pedro Chico Baptista. - 2026.

98 f. : il., color. ; 31 cm.

Orientador(a): Prof. Dr. Eduardo Luzeiro Feitosa.

Dissertação (mestrado) - Universidade Federal do Amazonas, Programa de Pós-Graduação em Informática, Manaus, 2026.

1. WebAssembly. 2. Cryptojacking. 3. Análise Estática 4. Aprendizado de Máquina. 5. Banco de dados vetorial.

I. Eduardo Luzeiro Feitosa. II. Universidade Federal do Amazonas. Programa de Pós-Graduação em Informática. III. Título.

Ministério da Educação
Universidade Federal do Amazonas
Coordenação do Programa de Pós-Graduação em Informática

FOLHA DE APROVAÇÃO

"IDENTIFICAÇÃO DE COMPORTAMENTO MALICIOSO EM WEBASSEMBLY "

BENTO PEDRO CHICO BAPTISTA

Dissertação de Mestrado defendida e aprovada pela banca examinadora constituída pelos Professores:

Prof. Dr. Eduardo Luzeiro Feitosa - **Presidente**

Prof. Dr. André Luiz da Costa Carvalho - **Membro Externo**

Dr. Hendrio Luis de Souza Bragança - **Membro Externo**

Documento assinado digitalmente
HENDRIO LUIS DE SOUZA BRAGANCA
Data: 11/05/2026 17:15:29-0300
Verifique em https://validar.iti.gov.br

Manaus, 24 de abril de 2026.

Documento assinado eletronicamente por **Eduardo Luzeiro Feitosa**, **Diretor em exercício**, em 24/04/2026, às 18:22, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **André Luiz da Costa Carvalho**, **Professor do Magistério Superior**, em 27/04/2026, às 13:09, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015 .

A autenticidade deste documento pode ser conferida no site https://sei.ufam.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **3131741** e o código CRC **80B43C96**.

Referência: Processo nº 23105.016670/2026-12 SEI nº 3131741

Dedico este trabalho à minha mãe e ao meu pai, que sempre me impulsionaram a perseguir meus sonhos e alcançar minhas metas. À minha esposa Alice e aos meus filhos Georgina, Haniel e Nelmo, cujo sacrifício compartilhado nos trouxe até aqui

Agradecimentos

Agradeço, em primeiro lugar, a N'Zambi, que iluminou meu caminho durante esta jornada, concedendo-me luz, força e sabedoria para que eu pudesse seguir até aqui. Tenho certeza de que, sem Ele, nada disso seria possível.

Agradeço aos meus amigos e colegas dos laboratórios Malware Hunter e IATS, em especial a Euler Viera e Andrés De Águas, pela parceria, pelo apoio e pelas incontáveis discussões e trocas de conhecimento. Sou grato por estarem sempre comigo, compartilhando momentos e me ajudando sempre que precisei. Vocês tornaram essa caminhada muito mais leve e especial.

Agradeço ao meu orientador, Professor Eduardo Feitosa, que me auxiliou em todos os momentos desta caminhada, sempre disposto a colaborar. Além de ser um excelente orientador, sempre me incentivou durante as orientações, acreditou em mim e confiou no meu trabalho. Obrigado, professor, pela paciência e pela amizade.

Agradeço aos membros da banca por aceitarem o convite para participar da defesa da minha dissertação. Escolhi vocês porque os enxergo como grandes referências e exemplos a serem seguidos.

Minha sincera gratidão a todos os professores do PPGI por compartilharem generosamente seus conhecimentos e por realizarem um trabalho de excelência que eleva a qualidade acadêmica da Universidade Federal do Amazonas. Agradeço igualmente ao IComp e a toda a equipe administrativa do PPGI, cujo suporte e infraestrutura foram essenciais para o desenvolvimento desta pesquisa. Registro meu agradecimento também à FAPEAM e à CAPES pelo apoio financeiro, sem o qual este trabalho não teria sido possível.

Acknowledgments

First and foremost, I thank N'Zambi, who illuminated my path throughout this journey, granting me light, strength, and wisdom so that I could reach this point. I am certain that, without Him, none of this would have been possible.

I thank my friends and colleagues from the Malware Hunter and IATS laboratories, especially Euler Viera and Andrés De Águas, for their partnership, support, and countless discussions and exchanges of knowledge. I am grateful for their constant presence, for sharing moments with me, and for helping me whenever I needed it. You all made this journey much lighter and more special.

I would like to thank my advisor, Professor Eduardo Feitosa, who has assisted me at every moment of this journey, always willing to collaborate. Beyond being an excellent advisor, he has always encouraged me during our meetings, believed in me, and trusted in my work. Thank you, Professor, for your patience and friendship.

I thank the members of the examining board for accepting the invitation to participate in the defense of my dissertation. I chose you because I see you as great references and examples to be followed.

My sincere gratitude to all the professors of the PPGI Program for generously sharing their knowledge and for carrying out work of excellence that elevates the academic quality of the Federal University of Amazonas. I also extend my thanks to IComp and the entire administrative team of PPGI, whose support and infrastructure were essential for the development of this research. I likewise acknowledge FAPEAM and CAPES for their financial support, without which this work would not have been possible.

"Em algum lugar, algo incrível está esperando para ser descoberto."

Carl Sagan

Resumo

O cryptojacking em ecossistemas web evoluiu drasticamente com a adoção do *WebAssembly* (*wasm*), cujos módulos binários de baixo nível facilitam a ofuscação algorítmica e a evasão de defesas tradicionais. Abordagens estáticas baseadas em assinaturas falham perante o metamorfismo sintático, enquanto a instrumentação dinâmica impõe sobrecargas operacionais intoleráveis. Para contornar estas vulnerabilidades, esta dissertação propõe uma arquitetura de análise estática suportada por classificação vetorial e fusão de características. O pipeline metodológico concebido intercepta e converte binários *wasm* em AST utilizando um parser desenvolvido em Rust, segmentando o código em unidades funcionais independentes. Em seguida, modelos de linguagem de grande escala geram *embeddings* que são matematicamente concatenados a métricas heurísticas determinísticas associadas a rotinas de *Proof-of-Work*. O vetor resultante operando em hiper-espaços de até 1027 dimensões é indexado em um Banco de Dados Vetorial Qdrant para inferência topológica. A avaliação empírica, conduzida sobre um corpus de 74 binários nativos (53.888 chunks), contrastou motores de decisão baseados em similaridade espacial (k-NN), consolidação estatística Machine Learning tradicional via *Mean Pooling*, e inferência cognitiva LLMs locais. Os resultados atestam que, enquanto a predição topológica atinge um *Mean Reciprocal Rank* de 0,9188 otimizada para triagem forense, a classificação agregada através de SVM Linear atinge um F1-Score de 0,9501, superando a ambiguidade semântica inerente a fragmentos isolados. Conclui-se que a projeção de código em espaços latentes proporciona uma identificação precisa e escalável de ameaças polimórficas, mitigando os custos da execução contínua.

Palavras-chave: WebAssembly, Cryptojacking, Análise Estática, Aprendizado de Máquina, Banco de Dados Vetorial, Processamento de Linguagem Natural, Fusão de Características.

Abstract

Cryptojacking in web ecosystems has evolved drastically with the adoption of *WebAssembly* (*Wasm*), whose low-level binary modules facilitate algorithmic obfuscation and evasion of traditional defenses. Signature-based static approaches fail in the face of syntactic metamorphism, while dynamic instrumentation imposes intolerable operational overhead. To circumvent these vulnerabilities, this dissertation proposes a static analysis architecture supported by vector classification and feature fusion. The methodological pipeline intercepts and converts Wasm binaries into an AST using a parser developed in Rust, segmenting the code into independent functional units. Subsequently, large language models generate embeddings that are mathematically concatenated with deterministic heuristic metrics associated with Proof-of-Work routines. The resulting vector, operating in hyperspaces of up to 1027 dimensions, is indexed in a Qdrant Vector Database for topological inference. The empirical evaluation, conducted on a corpus of 74 native binaries (53.888 chunks), evaluated and compared decision engine based on spatial similarity (k-NN), statistical consolidation via traditional Machine Learning (Mean Pooling), and cognitive inference from local LLMs. The results attest that, while topological prediction achieves a Mean Reciprocal Rank of 0.9188 optimized for forensic triage, aggregated classification via Linear SVM attains an F1-Score of 0.9501, surpassing the semantic ambiguity inherent to isolated fragments. It is concluded that the projection of code into hybrid latent spaces provides precise and scalable identification of polymorphic threats, mitigating the costs of continuous execution.

Keywords: WebAssembly, Cryptojacking, Static Analysis, Machine Learning, Vector Database, Natural Language Processing, Feature Fusion.

Lista de Figuras

2.1 Estrutura comparativa de assinaturas de funções (esquerda) e exportações/tabelas globais (direita) do binário WAT. 11
2.2 Fluxo de ataque de *cryptojacking*. 14
2.3 Fluxo de transformação de binários WebAssembly em representações estruturais utilizadas para extração de características e classificação automática de código. 19
2.4 Arquitetura do Transformer, Fonte: Vaswani et al. (2017). 26

4.1 Arquitetura do método proposto para detecção de *cryptojacking*. 49
4.2 Visita recursiva da AST em Rust com realce de sintaxe linear uniforme. 53

5.1 Matriz de confusão para o espaço vetorial Qwen. 66
5.2 Matriz de confusão para o espaço vetorial MPNet 66
5.3 Matrizes de confusão resultantes da validação cruzada para os modelos tradicionais. . . . 69
5.4 Matrizes de confusão para os modelos Llama 3.1 e 4 71
5.5 Visualização de dispersão dos *embeddings*, sendo os de cor azul legítimos e os de cor vermelho maliciosos. 73

Lista de Tabelas

2.1 Comparação entre os algoritmos de classificação quanto à interpretabilidade, custo computacional e principais limitações. 25
2.2 Comparação arquitetural entre os principais Sistemas de Gerenciamento de Bancos de Dados Vetoriais. 32

3.1 Definição do Objetivo de Revisão Conforme o Paradigma GQM. 36
3.2 Critérios de Inclusão (CI) e Exclusão (CE) considerados na Revisão Sistemática. 38
3.3 Resultados retornados da string de busca e progressão das fases de triagem. 38
3.4 Comparação estrutural e arquitetónica entre as ferramentas de detecção de *cryptojacking*. 45

4.1 Mapeamento da Arquitetura e Componentes de Software 50
4.2 Distribuição macroestrutural do conjunto de dados após aplicação de heurística de poda . 51
4.3 Configuração analítica dos classificadores baseline. 60

5.1 Especificações de hardware dos ambientes. 65
5.2 Métricas de desempenho preditivo do classificador k-NN ($k = 5$) no espaço de embeddings. 66
5.3 Métricas globais da validação cruzada 5-fold (Agregação de 74 Binários via Qwen). 68
5.4 Métricas globais de classificação ao nível do chunk para os modelos LLM. 70
5.5 Telemetria microarquitetural de busca k-NN sobre banco de grafos navegáveis (HNSW). . 74
5.6 Matriz comparativa de aplicabilidade dos motores de decisão. 75

Nomenclatura

AoT *Ahead-of-Time*

API *Application Programming Interface*

ASIC *Application-Specific Integrated Circuit*

AST *Abstract Syntax Tree*

C2 *Command and Control*

CFG *Control Flow Graph*

CNN *Convolutional Neural Network*

CPU *Central Processing Unit*

CUDA *Compute Unified Device Architecture*

DL *Deep Learning*

DOM *Document Object Model*

GPU *Graphics Processing Unit*

GQM *Goal, Question, Metric*

HNSW *Hierarchical Navigable Small World*

IA *Inteligência Artificial*

ICFG *Inter-Procedural Control Flow Graph*

IPS *Intrusion Prevention System*

IR *Intermediate Representation*

IRM *In-line Reference Monitor*

IVF *Inverted File Index*

JIT *Just-In-Time*

LIFO *Last-In, First-Out*

LLM *Large Language Model*

ML *Machine Learning*

NLP *Natural Language Processing*

PCA *Principal Component Analysis*

$PICOC$ *Population, Intervention, Comparison, Outcome, and Context*

PLN *Processamento de Linguagem Natural*

PoW *Proof-of-Work*

PQ *Pergunta de Pesquisa*

$RDBMS$ *Relational Database Management System*

SVM *Support Vector Machines*

$t-SNE$ *t-Distributed Stochastic Neighbor Embedding*

URI *Uniform Resource Identifier*

URL *Uniform Resource Locator*

$VDBMS$ *Vector Database Management System*

$VRAM$ *Video Random Access Memory*

$W3C$ *World Wide Web Consortium*

$WABT$ *WebAssembly Binary Toolkit*

WAT *WebAssembly Text Format*

$XGBoost$ *eXtreme Gradient Boosting*

Sumário

1 Introdução **2**

1.1 Motivação . . . 3

1.2 Hipótese e Questões de Pesquisa . . . 4

1.3 Objetivos . . . 4

1.4 Metodologia da Pesquisa . . . 5

1.5 Contribuições Esperadas . . . 5

1.6 Organização do Trabalho . . . 6

2 Fundamentos Básicos **7**

2.1 Webassembly . . . 7

2.1.1 Caraterísticas arquiteturais . . . 7

2.1.2 Modelo de segurança . . . 8

2.1.3 Estrutura Binária do Webassembly . . . 9

2.2 Ameaça do Cryptojacking . . . 10

2.2.1 Criptomineração no Contexto Web . . . 12

2.2.2 Adoção do WebAssembly em Campanhas Maliciosas . . . 13

2.3 Análise de Binários *WebAssembly* . . . 15

2.3.1 Representação Estrutural de Binários *WebAssembly* . . . 18

2.3.2 Extração de Features . . . 20

2.3.3 Heurísticas e Padrões Sintáticos Associados à Mineração . . . 21

2.4 Machine Learning . . . 22

2.4.1 Aprendizado Supervisionado . . . 23

2.4.2 Algoritmos de Classificação . . . 23

2.4.3 Deep Learning . . . 25

2.4.4 Arquitetura Transformers . . . 26

2.4.5 Chunking . . . 27

2.4.6 Representação Semântica . . . 27

2.4.7 Modelos de Linguagem de Grande Escala na Análise de Código . . . 28

2.5 Banco de Dados Vetoriais . . . 29

2.5.1 Métricas de Similaridade . . . 30

2.5.2 Algoritmos de Indexação . . . 31

2.6 Qdrant: Arquitetura e funcionamento . . . 31

2.6.1 Arquitetura de Indexação e Modelagem Espacial . . . 31

2.6.2 Posicionamento no Ecossistema de VDBMS . . . 32

2.6.3 Aplicações em Segurança da Informação . . . 33

2.7 Considerações Finais do Capítulo . . . 33

3 Trabalhos Relacionados **35**

3.1 Revisão Bibliográfica . . . 36
3.1.1 Protocolo . . . 36
3.1.2 Estratégia de Busca . . . 37
3.2 Ferramentas e Sistemas de Detecção . . . 43
3.3 Considerações Finais do Capítulo . . . 46

4 Método Proposto **48**
4.1 Visão Geral da Arquitetura . . . 48
4.2 Constituição e Curadoria do Conjunto de Dados . . . 50
4.2.1 Distribuição e Estatísticas Descritivas . . . 50
4.2.2 Critérios de Inclusão, Exclusão e Rotulagem . . . 51
4.3 Coleta de Binários WebAssembly . . . 52
4.3.1 Extração da AST com Rust . . . 52
4.3.2 Pré-processamento e Filtragem Heurística . . . 53
4.3.3 Isolamento Estrutural e Segmentação . . . 54
4.3.4 Extração de *Embeddings* . . . 56
4.3.5 Fusão de Features e Indexação de Características . . . 57
4.4 Classificação . . . 58
4.4.1 Classificação por Similaridade . . . 59
4.4.2 Classificação Baseline . . . 59
4.4.3 Classificação via Modelos de Linguagem de Grande Escala . . . 61
4.4.4 Auditoria e Visualização do Espaço Latente . . . 61
4.5 Métricas de Avaliação e Protocolo Experimental . . . 62
4.6 Considerações Finais do Capítulo . . . 63

5 Resultados **64**
5.1 Configurações e Ambiente Experimental . . . 64
5.2 Classificação por similaridade . . . 65
5.2.1 Implicações Práticas . . . 67
5.2.2 Acurácia versus precisão sob desbalanceamento de classes . . . 67
5.2.3 Implicações operacionais: tempo real versus análise forense . . . 68
5.3 Classificação Baseline . . . 68
5.4 Classificação via LLMs . . . 70
5.4.1 Latência de inferência e viabilidade de implantação . . . 72
5.5 Visualização Topológica . . . 72
5.6 Latência de inferência e análise de desempenho . . . 73
5.7 Discussão dos Achados e Análise Comparativa . . . 74
5.7.1 Granularidade da analise versus assertividade da decisão . . . 74
5.7.2 Análise Comparativa com o Estado da Arte . . . 75
5.7.3 Síntese de Mérito das Metodologias . . . 75
5.8 Considerações Finais do Capítulo . . . 75

6 Conclusão **77**
6.1 Desafios . . . 78
6.2 Contribuições Alcançadas . . . 78
6.3 Trabalhos Futuros . . . 79

Referências Bibliográficas **80**

Capítulo 1

Introdução

Cryptojacking (mineração não autorizada de criptomoedas) consolidou-se como uma ameaça emergente e relevante no campo da segurança da informação, especialmente em ambientes web. Desde 2017, com a introdução e popularização do Coinhive [1], observou-se um crescimento significativo na adoção dessa prática por agentes maliciosos. De acordo com a Cyber Threat Alliance (2018)[2], houve um aumento de 459% nos ataques de *cryptojacking* no ano subsequente ao lançamento do Coinhive. Relatórios recentes indicam que, em 2023, os ataques voltaram a crescer, com um incremento de 400% apenas no primeiro semestre, em comparação ao mesmo período do ano anterior (Sonic Wall, 2024) [3]. Um exemplo concreto dessa tendência foi documentada pelos analistas da Cside [4], que relataram a descoberta de uma campanha de *cryptojacking* que infectou mais de 3.500 websites (Cside, 2025), demonstrando que essa modalidade de ataque continua sendo uma ameaça relevante.

Embora técnicas tradicionais de detecção de *cryptojacking* sejam úteis como primeira linha de triagem, sua eficácia é limitada quando confrontadas com ambientes Web heterogêneos e adversários adaptativos. Abordagens baseadas em assinatura dependem de padrões estáticos e, portanto, são facilmente contornadas por empacotamento, ofuscação, diversificação estrutural e pequenas mutações no código. Medidas de anomalia por consumo de CPU (*Central Processing Unit*) incorrem em falsos negativos quando o agente malicioso aplica *throttling*, distribuição de carga em *Web Workers* ou ativação intermitente condicionada a eventos. Monitores de comportamento em tempo de execução, por sua vez, sofrem com ruído de aplicações legítimas intensivas em computação e com o custo de instrumentação, o que reduz sua aplicabilidade em larga escala. Em todos esses casos, a superfície de evasão permanece ampla e se beneficia do fato de que os indicadores observados não capturam, de forma direta, a finalidade semântica do programa suspeito (Konoth et al., 2018; Hong et al., 2018; Harnes and Morrison, 2024b).

Para agravar este cenário, o advento do *WebAssembly* (*Wasm*) introduziu novas implicações de segurança no cenário de *cryptojacking*. Por definir um formato binário portátil e de baixo nível, projetado para execução com desempenho próximo ao nativo em navegadores modernos, *wasm* passou a ser amplamente adotado por agentes maliciosos em campanhas de mineração não autorizada de criptomoedas (Harnes and Morrison, 2024a,b).

Esse cenário motiva a adoção de mecanismos que incorporem uma análise mais profunda do código, buscando sinais estáveis do comportamento minerador que resistam a técnicas de evasão. Em vez de depender exclusivamente de assinaturas rígidas ou limiares dinâmicos de uso de recursos, tais mecanismos

[1] Coinhive foi um serviço que fornecia um script para proprietários de sites minerarem a criptomoeda Monero, utilizando o poder de processamento da CPU dos visitantes.

[2] Cyber Threat Alliance (CTA) é uma organização sem fins lucrativos (501(c)(6)) que visa melhorar a segurança cibernética do ecossistema digital global.

[3] Sonic Wall é uma empresa líder em cibersegurança, especializada no desenvolvimento de soluções de segurança de rede.

[4] A Cside é uma empresa de cibersegurança que fornece soluções de segurança do lado do cliente orientadas por IA para detetar e prevenir scripts maliciosos de terceiros e fraudes.

devem ser capazes de extrair a lógica subjacente do binário. Ao elevar o nível de abstração para representações intermediárias como a AST *(Árvore de Sintaxe Abstrata)*, torna-se possível mapear a organização estrutural de laços de repetição e operações aritméticas intensivas, reduzindo a sensibilidade a variações cosméticas e ofuscações. Esses fundamentos abrem caminho para o uso inovador de LLMs *(Modelos de Linguagem)* não como classificadores diretos, mas como potentes motores de extração de representações semânticas *(embeddings*[5]*)*. Quando essas representações latentes são combinadas com artefatos sintáticos explícitos, cria-se um modelo robusto para a detecção de mineração em navegadores (Harnes and Morrison, 2024b).

1.1 Motivação

O crescimento do *cryptojacking* como técnica de exploração silenciosa de navegadores representa um desafio persistente à segurança de sistemas web. Por sua natureza furtiva e baixo custo operacional, essa prática tornou-se um vetor atrativo para agentes maliciosos, especialmente em contextos onde a detecção é dificultada por limitações técnicas ou políticas permissivas de execução de código. A crescente adoção do *WebAssembly* agrava esse cenário, uma vez que seu alto desempenho e portabilidade são ideias para serem exploradas como plataforma de execução para mineradores ofuscados e evasivos, dificultando a inspeção sintática e desafiando mecanismos tradicionais de detecção.

Ferramentas de detecção de criptomineradores escritos em *wasm* como o MinerSweeper (Konoth et al., 2018), que identifica assinaturas do algoritmo *CryptoNight*[6] em binários *Wasm*, e o MinerRay (Romano et al., 2020), que utiliza CFG (*Control Flow Graph*) para modelar padrões semânticos associados à mineração, apresentam bons resultados, mas também enfrentam desafios frente a técnicas de evasão mais recentes e variantes ofuscadas (Romano et al., 2020; Kim et al., 2022). Por exemplo, o MinerRay adota uma abordagem de análise semântica, convertendo JavaScript e *WebAssembly* para uma representação intermediária e construindo CFGs capazes de modelar padrões típicos de algoritmos de *hashing*. Essa estratégia permite detectar mineração mesmo em variantes e linguagens distintas, oferecendo maior robustez do que métodos baseados em assinaturas ou métricas de uso de CPU. Ainda assim, enfrentam evasões recentes por exemplo, ofuscação agressiva, encriptação de blocos de código.

Abordagens guiadas por semântica tendem a ser menos sensíveis a ofuscações sintáticas, embora técnicas de cifragem integral do código ainda prejudiquem métodos estáticos. É por isso que as LLMs têm sido explorados para melhorar a descompilação e a compreensão de *wasm* (Romano et al., 2020; She et al., 2024), embora sua resiliência a ofuscação avançada em cenários de detecção de *cryptojacking* permanece não estabelecida.

LLMs também já foram utilizadas para extrair estruturas semânticas de binários *WebAssembly*, como no WaDec (She et al., 2024). Isso sustenta a premissa de que a representação textual de AST carrega assinaturas latentes de comportamento. Contudo, o treinamento ponta a ponta *end-to-end* de LLMs para a tarefa direta de classificação de malware enfrenta desafios severos, como o alto custo computacional, a dificuldade em processar o volume massivo de tokens gerados por binários extensos e a opacidade inerente natureza black-box das decisões tomadas por redes neurais profundas.

Diante desses cenários, este trabalho propõe uma mudança de paradigma: a transição de um modelo puramente gerativo/classificatório para uma abordagem de recuperação e busca vetorial híbrida. Em vez de depender exclusivamente da LLM para ditar se um código é malicioso, a pesquisa explora a capacidade desses modelos em atuar como extratores de *embeddings* semânticos densos a partir dos *chunks* da

[5]Embeddings são representações vetoriais contínuas de objetos discretos (palavras, itens) que codificam similaridade semântica pela proximidade espacial (Mikolov et al., 2013b).

[6]CryptoNight é um algoritmo de prova de trabalho introduzido como parte do protocolo CryptoNote (Feng and Luo, 2020).

AST. Para contornar a evasão por ofuscação, essas representações semânticas profundas são integradas a características estruturais determinísticas e explícitas como a densidade de instruções I64, operações de cifragem *I64Xor* e ciclos de repetição *Loop* mapeadas durante a fase de pré-processamento. Essa fusão gera um espaço vetorial capaz de representar tanto a "intenção" do código semântica quanto a sua "mecânica"operacional heurística. A classificação ocorre, então, pela análise de vizinhança e similaridade neste espaço vetorial.

Assim, a pesquisa visa desenvolver mecanismos de defesa contra o *cryptojacking* que sejam não apenas robustos e resilientes a ofuscações, mas também escaláveis e interpretáveis, superando as fragilidades das abordagens estáticas tradicionais e dos modelos puramente neurais

1.2 Hipótese e Questões de Pesquisa

Parte-se da hipótese de que a integração de funções de *features* semânticas latentes com caraterísticas heurísticas sintáticas de AST em um espaço de busca vetorial, permite a identificação precisa de *cryptojacking* em *WebAssembly*, reduzindo as taxas de erro e demonstrando maior eficácia a técnicas de ofuscação.

Para avaliar essa hipótese, propõem-se as seguintes questões de pesquisa:

- **QP-1.** Quais características estruturais e heurísticas na AST de binários *WebAssembly* (como operações aritméticas de 64 bits e estruturas de controle de fluxo) apresentam a maior correlação com atividades de mineração intensiva?

- **QP-2.** De que forma a fusão *features* de vectores de *embeddings* semânticos com características determinísticas impacta o agrupamento entre amostras benignas e maliciosas no espaço vetorial?

1.3 Objetivos

O objetivo geral desta pesquisa é desenvolver e avaliar um método para a detecção de *cryptojacking* em aplicações *WebAssembly*, visando superar as limitações das abordagens estáticas tradicionais por meio de uma arquitetura que combina a geração de representações semânticas e a extração de características sintáticas da AST, operacionalizando a classificação.

Para alcançar esse objetivo, foram definidos os seguintes objetivos específicos:

i. Caracterizar a estrutura e o comportamento de criptomineradores baseados em WebAssembly, identificando padrões sintáticos e estruturais associados às rotinas de mineração.

ii. Construir um conjunto de dados rotulado, composto por binários WebAssembly benignos e maliciosos, que sirva de base empírica para as avaliações.

iii. Propor uma representação de código que combine caraterísticas heurísticas determinísticas (como a densidade de operações aritméticas de 64 bits e a frequência de ciclos de repetição) com representações semânticas latentes.

iv. Avaliar a capacidade de discriminação do método proposto entre as amostras benignas e maliciosas, por meio de métricas quantitativas de classificação.

1.4 Metodologia da Pesquisa

Enquanto os objetivos específicos delimitam os resultados pretendidos, esta seção descreve o percurso metodológica para alcançá-los. A abordagem está estruturada em quatro etapas sequenciais e complementares, incluíndo a montagem do cenário experimental de avaliação

1. **Revisão bibliográfica e coleta de dados**: Realização de uma revisão da literatura sobre técnicas de detecção de *cryptojacking* em navegadores que utilizam *WebAssembly*, com o objetivo de identificar abordagens existentes, lacunas metodológicas e oportunidades de aprimoramento. Em complemento, serão selecionados *datasets* públicos e coletados binários *WebAssembly* de repositórios e *websites*, visando compor um conjunto de dados representativo para experimentação e validação da proposta.

2. **Extração sintática e engenharia de caraterística**: Os arquivos .wasm obtidos na etapa anterior serão processados em baixo nível para extração de suas AST. Em seguida, aplicar-se-ão filtros heurísticos para segmentar o código em funções individuais e quantificar anomalias sintáticas que denotem computação intensiva.

3. **Geração de representações e indexação vetorial**: Nesta etapa, as representações textuais da AST segmentada serão submetidas a um LLM pré-treinado para a extração de *embeddings* semânticos. Esses vetores densos serão matematicamente concatenados aos metadados heurísticos extraídos na etapa anterior, e o vetor resultante será indexado em um Banco de Dados Vetorial, criando um espaço multidimensional de representação de código.

4. **Avaliação experimental**: com o modelo treinado, serão conduzidos experimentos controlados para validar a abordagem proposta. Serão aplicadas métricas quantitativas e qualitativas, como acurácia, precisão e *recall*, além de métricas estatísticas (covariância, coeficiente de variação), e tempo médio de inferência, para mensurar o desempenho do método de detecção. Além disso, serão realizadas análises qualitativas visando à interpretação dos resultados e à identificação de padrões indicativos de scripts maliciosos.

1.5 Contribuições Esperadas

As contribuições esperadas ao final desta pesquisa incluem:

- Extração de metadados estruturais referentes a chamadas de sistema e controle de tempo no nível do *bytecode*, fornecendo enriquecimento contextual heurístico para diferenciar rotinas lícitas de alta performance de campanhas ofuscadas de *cryptojacking*.

- Criação de uma Prova de Conceito baseada no método proposto, capaz de identificar cripto mineradores em binários *WebAssembly* aplicando fusão de *features* em *embeddings* em um banco dados vetorial.

- Avaliação de uma prova de conceito, por meio de métricas quantitativas e análises qualitativas, a fim de validar sua eficácia em cenários reais de *cryptojacking*.

- Disponibilização de um conjunto de dados contendo amostras benignas e maliciosas de binários *wasm*, suas respectivas representações em AST segmentada e os vetores de métricas extraídas, visando à reprodutibilidade da pesquisa e ao fortalecimento da base empírica da comunidade de segurança cibernética.

1.6 Organização do Trabalho

Este documento está organizado em cinco capítulos, incluindo esta Introdução. O Capítulo 2 apresenta os conceitos necessários para a compreensão da pesquisa. O Capítulo 3 descreve os trabalhos relacionados sobre detecção de *cryptojacking*. O Capítulo 4 apresenta a proposta desta pesquisa. O Capítulo 5 apresenta a avaliação dos resultados por último Capitulo 6 as conclusões e os trabalhos futuros.

Capítulo 2

Fundamentos Básicos

Este capítulo apresenta os conceitos fundamentais que sustentam a proposta desta dissertação. Abordam-se os princípios operacionais do *WebAssembly*, os mecanismos associados à *cryptojacking* e os principais métodos de análise empregados na detecção de binários *WebAssembly* utilizados nesse tipo de ameaça.

2.1 Webassembly

WebAssembly (Wasm) é um formato binário de instruções projetado para execução em uma máquina virtual baseada em pilha LIFO *(Last In First Out)* (Hoffman, 2019; Heinrich et al., 2024). A tecnologia foi concebida como um alvo de compilação portátil para linguagens de programação de alto nível, incluindo C, C++ e Rust, viabilizando sua implementação na web tanto para aplicações no lado do cliente quanto do lado do servidor e em outros ambientes computacionais (Haas et al., 2017; W3C, 2024).

A especificação do *WebAssembly* foi desenvolvida de forma colaborativa em 2017 por empresas da Mozilla, Google, Apple e Microsoft, tendo como objetivo ampliar as capacidades computacionais da web. Diferentemente de linguagens tradicionalmente utilizadas em navegadores, como o JavaScript, o *wasm* utiliza um formato binário compacto que pode ser rapidamente validado e executado pelos motores de execução dos navegadores, reduzindo a sobrecarga associada à interpretação de código e proporcionado desempenho próximo ao de aplicações nativas (Haas et al., 2017).

No ecossistema web, o *wasm* não foi projetado para substituir o JavaScript, mas para atuar de forma complementar (Hoffman, 2019; Golsch, 2019). Nesse modelo, rotinas computacionalmente intensivas podem ser implementadas em linguagens compiladas e executadas como módulos *wasm*, enquanto o JavaScrit permanece responsável pela interação com o DOM *(Document Object Model)*, manipulação de interface de acesso a APIs *(Application Programming Interface)* do navegador.

2.1.1 Caraterísticas arquiteturais

O *WebAssembly* apresenta um conjunto de propriedades que favorecem sua adoção no desenvolvimento de aplicações web de alto desempenho. Caraterizado pela portabilidade o binário *wasm* foi projetado para ser independente de arquitetura de hardware e sistema operacional, permitindo que um mesmo módulo binário seja executado em diferentes ambientes que implementem a máquina virtual *WebAssembly*, outra característica relevante é o desempenho próximo ao nativo. Como o código *wasm* é produzido por compiladores tradicionais, muitas otimizações de baixo nível são realizadas durante o processo de compilação, reduzindo o custo de execução em comparação com linguagens interpretadas. O *WebAssembly* também utiliza um formato binário compacto `.wasm`, que reduz o tamanho dos artefatos transferidos pela rede e melhora o tempo de carregamento das aplicações e para fins de análise e depuração tem a

representação textual `.wat` (*Webassembly Text Format*).

Outra propriedade importante é o modelo de memória linear, no qual cada módulo possui um espaço de memória contínuo e isolado. Esse modelo simplifica a execução de programas compilados e facilita a interoperabilidade com linguagens que utilizam memória contígua. Além disso, o *WebAssembly* possui interoperabilidade direta com JavaScript, permitindo que funções e estruturas de dados sejam compartilhadas entre módulos *wasm* e scripts JavaScript. Essa integração permite combinar a flexibilidade do JavaScript com o desempenho computacional de código compilado.

2.1.2 Modelo de segurança

O modelo de segurança do *WebAssembly* foi concebido para garantir que módulos executados no navegador operem de forma isolada e controlada. Esse modelo baseia-se principalmente em dois mecanismos: validação estrutural do código e execução em ambiente *sandboxed*. Antes da execução, os módulos *wasm* passam por um processo de validação estática, no qual o motor de execução verifica se o código atende às restrições definidas pela especificação. Esse processo garante que o fluxo de controle seja bem formado e que as operações sobre memória e tipos sejam consistentes. Além disso, os módulos são executados em um ambiente *sandbox*, que restringe o acesso direto a recursos do sistema operacional. A interação com o ambiente externo ocorre exclusivamente por meio de interfaces explícitas definidas pelo ambiente hospedeiro, geralmente o navegador. Embora esses mecanismos reduzam diversas classes de falhas comuns em software nativo, eles não eliminam completamente os riscos de segurança associados ao uso da tecnologia.

Vulnerabilidade e vetores de ataques

Apesar das propriedades de isolamento e validação presentes na especificação, o *WebAssembly* pode introduzir novos desafios de segurança. Uma análise sistemática recente conduzida por Perrone and Romano (2025) identifica diferentes classes de ameaças associadas ao ecossistema *WebAssembly*, relacionadas principalmente à interação entre módulos *wasm*, motores de execução e componentes do ambiente hospedeiro.

Uma das principais fontes de risco decorre das vulnerabilidades herdadas das linguagens de origem. Como muitos módulos *wasm* são gerados a partir de linguagens com gerenciamento manual de memória, como C e C++, vulnerabilidades clássicas, incluindo estouros de *buffer* e corrupção de memória, podem ser preservadas durante o processo de compilação. Outra classe relevante corresponde aos ataques de canal lateral, nos quais informações sensíveis podem ser inferidas a partir de características temporais ou microarquiteturas da execução. Mesmo em ambientes *sandboxed*, medições de tempo e padrões de acesso à cache podem revelar informações relacionadas ao comportamento interno do programa.

A integração entre *WebAssembly* e JavaScript também cria uma superfície adicional de ataque. Interfaces de importação e exportação entre módulos podem ser utilizadas para carregar dinamicamente código malicioso ou manipular estruturas internas da aplicação. A natureza binária do formato *wasm* também favorece o uso de técnicas de ofuscação e evasão de análise, nas quais transformações estruturais são aplicadas ao *bytecode* com o objetivo de dificultar a análise estática ou dinâmica do código executado.

Além disso, o desempenho elevado proporcionado pelo *WebAssembly* permite a execução eficiente de tarefas computacionalmente intensivas diretamente no navegador. Essa característica tem sido explorada em ataques de *cryptojacking*, nos quais módulos *wasm* são utilizados para executar algoritmos de mineração de criptomoedas utilizando os recursos computacionais do usuário sem seu consentimento (Konoth et al., 2018). Estudos indicam que implementações de mineração em navegadores frequentemente utilizam combinações de JavaScript e *WebAssembly* para maximizar o desempenho e dificultar mecanismos de detecção baseados em assinaturas ou análise superficial de scripts (Romano et al., 2020).

Diante desse cenário, a análise estrutural e comportamental de módulos *WebAssembly* torna-se um elemento importante para o desenvolvimento de mecanismos capazes de identificar atividades maliciosas no ambiente web. Nesse contexto, compreender a organização interna dos binários *wasm* e suas propriedades de execução constitui um passo fundamental para o estudo de técnicas de detecção e mitigação de ameaças associadas a essa tecnologia.

A ofuscação em módulos *WebAssembly* tem sido objeto de investigações recentes, embora ainda sejam pouco comum transformações realizadas diretamente no nível binário. Em geral, os indícios de ofuscação observados não revelam modificações explícitas aplicadas ao código de máquina. Contudo, esse cenário começa a se alterar. A técnica descrita por Cabrera-Arteaga et al. (2023) demonstra a possibilidade de aplicar transformações diretamente no *bytecode* por meio da ferramenta *wasm-mutate*, permitindo obscurecer a estrutura do código sem comprometer sua funcionalidade. De forma complementar, She et al. (2024) mostram que modificações binárias no *wasm* podem produzir efeitos similares à ofuscação tradicional, mesmo quando empregadas com finalidades distintas, como a evasão de mecanismos de detecção.

Diante da versatilidade e do elevado desempenho proporcionados pelo *WebAssembly*, bem como de sua crescente suscetibilidade à exploração maliciosa, evidencia-se a necessidade de abordagens analíticas mais precisas e abrangentes para sua inspeção. Técnicas de evasão, como a diversificação automática de binários, têm demonstrado a capacidade de variantes de *cryptojacking* em contornar mecanismos tradicionais de detecção, incluindo métodos baseados em aprendizado de máquina e análise estática (Cabrera-Arteaga et al., 2023). Esse cenário reforça a importância de mecanismos que integrem análise estrutural e semântica na identificação de comportamentos maliciosos em binários *wasm*. Na sequência, são detalhadas a estrutura dos binários *wasm*, sua relação com práticas de *cryptojacking* e as técnicas utilizadas para sua análise.

2.1.3 Estrutura Binária do Webassembly

O *WebAssembly* foi projetado para utilizar um formato binário compacto, otimizado para transmissão eficiente, validação rápida e compilação progressiva em navegadores e outros ambientes de execução (Haas et al., 2017; Wang et al., 2018). Esse formato permite que módulos sejam carregados e processados de maneira incremental, reduzindo a latência de inicialização e favorecendo a execução de aplicações de alto desempenho. Internamente, o código *WebAssembly* é organizado em módulos binários estruturados, nos quais diferentes componentes do programa são definidos em seções distintas. Essa organização modular facilita tanto a validação estrutural quanto a compilação do código durante o carregamento do módulo (Ray, 2023).

Cada módulo constitui a unidade fundamental de distribuição e execução no ecossistema *WebAssembly*. Do ponto de vista estrutural, um módulo contém declarações explícitas de tipos, funções, memórias, tabelas e variáveis globais, além de informações sobre importações e exportações que permitem a interação com o ambiente hospedeiro, geralmente implementado em JavaScript no contexto de navegadores web (Heinrich et al., 2024).

O arquivo binário inicia-se com um preâmbulo, seguido por uma sequência de seções que descrevem os diferentes componentes do módulo. O preâmbulo consiste em uma sequência de *bytes* responsável por identificar o arquivo como um módulo *WebAssembly* válido e indicar sua versão (Kim et al., 2022).

Após o preâmbulo, o módulo apresenta um conjunto de seções padronizadas que contêm as informações necessárias para a validação e execução do código. Entre as seções mais relevantes destacam-se:

- ***Type Section***: declara as assinaturas de funções utilizadas no módulo, especificando tipos de parâmetros e valores de retorno.

- ***Import Section***: lista dependências externas necessárias à execução do módulo, incluindo funções, memórias ou variáveis globais fornecidas pelo ambiente hospedeiro.
- ***Function Section***: associa as funções definidas no módulo às assinaturas previamente declaradas na seção de tipos.
- ***Table Section***: define tabelas que armazenam referências a funções, permitindo chamadas indiretas durante a execução.
- ***Memory Section***: declara a memória linear do módulo, especificando seus limites iniciais e máximos.
- ***Global Section***: registra variáveis globais utilizadas pelas funções do módulo.
- ***Export Section***: torna determinados elementos do módulo acessíveis ao ambiente hospedeiro.
- ***Code Section***: contém o corpo das funções definidas no módulo, codificado em bytecode WebAssembly.

Além das seções padronizadas, a especificação também permite a inclusão de seções personalizadas (*Custom Sections*), que podem armazenar informações auxiliares como nomes simbólicos de funções ou metadados de depuração. Essas seções não são necessárias para a execução do módulo e são geralmente ignoradas durante o processo de validação.

A Figura **??** apresenta um exemplo da representação textual de um módulo WebAssembly no formato `.wat`. Nessa representação é possível observar a organização declarativa dos principais componentes do módulo, incluindo definições de funções, exportações e declarações de memória.

A estrutura declarativa dos módulos *WebAssembly* facilita a análise estrutural do binário, permitindo que ferramentas especializadas examinem individualmente cada componente do módulo. Essa propriedade tem sido explorada em estudos de análise estática e dinâmica de código *WebAssembly*, possibilitando a extração de características estruturais e comportamentais relevantes para identificação de padrões de execução e investigação de atividades potencialmente maliciosas.

No contexto desta pesquisa, a compreensão da organização interna dos módulos *wasm* constitui um passo fundamental para a análise e inspeção de binários utilizados em aplicações web, especialmente em cenários que envolvem o uso de *WebAssembly* para execução de cargas computacionais intensivas.

2.2 Ameaça do Cryptojacking

A criptomineração é o processo pelo qual novas unidades de criptomoedas são geradas e transações existentes são validadas em redes distribuídas. Esse processo envolve a resolução de problemas matemáticos complexos que demandam intensa capacidade computacional, com destaque para o uso de funções de *hash* (Konoth et al., 2018; Wang et al., 2018; Romano et al., 2020; Naseem et al., 2021). As criptomoedas, por sua vez, constituem ativos digitais descentralizados, concebidos para operar e serem transacionados em ambientes virtuais, baseando-se em tecnologias de registro distribuído, como a *blockchain* [1] (Konoth et al., 2018; Kaza et al., 2024; Coinbase, 2025).

Existem duas formas principais de se obter criptomoedas: por meio da aquisição direta em mercados especializados ou por meio de mineração. No segundo caso, os participantes que contribuem com poder computacional para resolver os desafios criptográficos são recompensados com novas moedas. Contudo,

[1] Blockchain é uma estrutura de dados de registro distribuído e imutável, que utiliza encadeamento criptográfico para garantir a integridade e a ordem cronológica de transações em uma rede descentralizada.

```
(module
  (type (;0;) (func (param i32 i32 i32)))
  (type (;1;) (func (param i32 i32 i32) (result i32)))
  (type (;2;) (func (param i32) (result i32)))
  (type (;3;) (func (result i32)))
  (type (;4;) (func (param i32)))
  (type (;5;) (func (param i32 i32 i32 i32)))
  (type (;6;) (func (param i32 i32) (result i32)))
  (type (;7;) (func (param i32 i32)))
  (type (;8;) (func (param i32 i32 i64)))
  (type (;9;) (func (param i32 i64 i32)))
  (type (;10;) (func (param i32 i32 i64 i32) (result i32
    )))
  (type (;11;) (func (param i32 i32 i32 i32) (result i32
    )))
  (import "env" "memory" (memory (;0;) 256))
  (import "env" "table" (table (;0;) 32 32 funcref))
  (import "env" "tableBase" (global (;0;) i32))
  (import "env" "DYNAMICTOP_PTR" (global (;1;) i32))
  (import "env" "STACKTOP" (global (;2;) i32))
  (import "env" "STACK_MAX" (global (;3;) i32))
  (import "env" "enlargeMemory" (func (;0;) (type 3)))
  (import "env" "getTotalMemory" (func (;1;) (type 3)))
  (import "env" "abortOnCannotGrowMemory" (func (;2;) (
    type 3)))
  (import "env" "abortStackOverflow" (func (;3;) (type
    4)))
  (import "env" "nullFunc_ii" (func (;4;) (type 4)))
  (import "env" "nullFunc_iiii" (func (;5;) (type 4)))
  (import "env" "nullFunc_viii" (func (;6;) (type 4)))
  (import "env" "___assert_fail" (func (;7;) (type 5)))
  (import "env" "___lock" (func (;8;) (type 4)))
  (import "env" "___setErrNo" (func (;9;) (type 4)))
  (import "env" "___syscall140" (func (;10;) (type 6)))
  (import "env" "___syscall146" (func (;11;) (type 6)))
  (import "env" "___syscall20" (func (;12;) (type 6)))
  (import "env" "___syscall54" (func (;13;) (type 6)))
  (import "env" "___syscall6" (func (;14;) (type 6)))
  (import "env" "___unlock" (func (;15;) (type 4)))
  (import "env" "___exit" (func (;16;) (type 4)))
  (import "env" "_emscripten_memcpy_big" (func (;17;) (
    type 1)))
  (import "env" "_ftime" (func (;18;) (type 2)))
  (import "env" "_gtime" (func (;19;) (type 2)))
  (func (;20;) (type 1) (param i32 i32 i32) (result i32)
    (local i32 i32 i32)
    local.get 2
    i32.const 8192
    i32.ge_s
    if ;; label = @1
      local.get 0
      local.get 1
      local.get 2
      call 17
      return
    end
    local.get 0
    local.get 4
```

```
    i32.store offset=8
    local.get 1
    local.get 8
    i32.store offset=12)
  (func (;71;) (type 0) (param i32 i32 i32) ...
  (func (;72;) (type 0) (param i32 i32 i32) ...
  (func (;73;) (type 0) (param i32 i32 i32) ...
  (func (;74;) (type 2) (param i32) (result i32) ...
  (global (;4;) (mut i32) (global.get 1))
  (global (;5;) (mut i32) (global.get 2))
  (global (;6;) (mut i32) (global.get 3))
  (global (;7;) (mut i32) (i32.const 0))
  (global (;8;) (mut i32) (i32.const 0))
  (global (;9;) (mut i32) (i32.const 0))
  (export "____errno_location" (func 54))
  (export "_cryptonight_hash" (func 46))
  (export "_fflush" (func 40))
  (export "_free" (func 24))
  (export "_malloc" (func 43))
  (export "_memmove" (func 39))
  (export "_sbrk" (func 26))
  (export "establishStackSpace" (func 56))
  (export "getTempRet0" (func 48))
  (export "setTempRet0" (func 49))
  (export "setThrew" (func 50))
  (export "stackAlloc" (func 74))
  (export "stackRestore" (func 62))
  (export "stackSave" (func 66))
  (elem (;0;) (global.get 0) func 28 28 28 28 28 57 28
    28 23 23 23 23 23 23 42 55 53 23 23 23 23 23 23 23
    35 73 47 72 71 35 35 35)
  (data (;0;) (i32.const 1024) "\c6cc\a5\f8||\84\eeww
    \99\f6{{\8d\ff\f2\f2\0d\d6kk\bd\deoo\b1\91\c5\
    c5T`00P\02\01\01\03\cegg\a9V++}")
```

Figura 2.1: Estrutura comparativa de assinaturas de funções (esquerda) e exportações/tabelas globais (direita) do binário WAT.

o processo de mineração demanda considerável investimento financeiro e conhecimento técnico, o que o torna inacessível para a maioria dos usuários comuns (Varlioglu et al., 2020).

Historicamente, a mineração de criptomoedas teve início utilizando CPUs de computadores domésticos, seguindo o princípio original de descentralização da rede. Contudo, a crescente dificuldade computacional impulsionou a transição para sistemas de hardware especializado, como as GPUs (*Graphics Processing*

Unit) e, posteriormente, os ASICs (*Application-Specific Integrated Circuit*), que oferecem maior eficiência energética e poder de processamento (Taylor, 2017).

No entanto, esse cenário de especialização começou a ser desafiado por criptomoedas projetadas especificamente para serem resistentes a ASICs. Moedas como Monero[2], Bytecoin[3] e Electroneum[4] utilizam algoritmos que limitam a vantagem de dispositivos especializados ao exigir um uso intensivo de memória, visando uma distribuição mais equitativa do poder de mineração (Musch et al., 2019). Paradoxalmente, essa resistência ao hardware dedicado permitiu o retorno da mineração às CPUs Hong et al. (2018), mas sob uma nova faceta: o aproveitamento de tecnologias web modernas para a mineração diretamente no navegador, sem a necessidade de instalação de software adicional (Romano et al., 2020). Essa viabilidade técnica deu origem a campanhas de *cryptojacking*, onde o poder computacional do usuário é explorado de forma maliciosa através de scripts integrados em websites (Konoth et al., 2018).

O fenômeno do *cryptojacking* ganhou notoriedade a partir de 2017, impulsionado pela introdução da plataforma Coinhive, voltada para a mineração de Monero em navegadores. Nesse período, observou-se um aumento de até 600% nos incidentes relacionados (Wang et al., 2018), em um contexto de declínio dos ataques tradicionais de *ransomware*, cujo impacto foi reduzido pela crescente resistência das vítimas em atender às exigências de pagamento (Lonkar and Chandrayan, 2018). Explorando algoritmos como o *CryptoNight* e o *RandomX*, agentes maliciosos passaram a incorporar *WebAssembly* como vetor de ataque, utilizando os recursos computacionais da vítima para minerar cripto moedas em tempo real, sem seu consentimento (Konoth et al., 2018; Lonkar and Chandrayan, 2018; Kaza et al., 2024).

2.2.1 Criptomineração no Contexto Web

A proximidade de desempenho entre o *WebAssembly* e o código nativo tem tornado essa tecnologia atrativa para agentes maliciosos, especialmente em campanhas de *cryptojacking*. Nesses ataques, módulos `.wasm` são empregados para minerar cripto moedas diretamente no navegador, aproveitando a eficiência computacional, o isolamento de memória e a ausência de coleta de lixo que caracterizam o modelo de execução do *webassembly* (Lonkar and Chandrayan, 2018; Konoth et al., 2018). Como relatado por Romano et al. (2020), tais atributos favorecem o uso do *WebAssembly* em operações intensivas, como a mineração de Monero, e ampliam sua adoção por mineradores modernos.

Relatórios apontam que, apenas um ano após a introdução do *WebAssembly*, os ataques de *cryptojacking* envolvendo essa tecnologia aumentaram em 459% (Cyber Threat Alliance, 2018). No ano seguinte, constatou-se que aproximadamente metade dos sites que utilizavam módulos *webassembly* estava envolvida em ataques de *cryptojacking* (Harnes and Morrison, 2024b). Essa tendência evidencia a dificuldade na detecção desses binários por ferramentas tradicionais baseadas em assinaturas, devido à sua representação compacta e ausência de padrões legíveis (Romano et al., 2020; Lonkar and Chandrayan, 2018).

Além disso, algoritmos como *CryptoNight* e *RandomX*, utilizados predominantemente na mineração de Monero, são favorecidos pelo *webassembly* por exigirem grande volume de operações de ponto flutuante e acesso intensivo à memória, características que se beneficiam do modelo baseado em pilha e da execução eficiente do *wasm* (Konoth et al., 2018; Kaza et al., 2024). Muitos mineradores também adotam estratégias de *throttling* de CPU, ajustando dinamicamente o uso de recursos para permanecerem abaixo dos limiares de detecção comportamental (Sonic Wall, 2024).

A migração da mineração em JavaScript para o *webassembly* ampliou significativamente o desempenho das rotinas de mineração executadas no navegador, permitindo taxas mais elevadas de cálculo de *hashes* por segundo. Além do ganho de desempenho, a representação binária compacta do *wasm* e a dificuldade de inspeção direta do código contribuem para aumentar a furtividade dessas campanhas, dificultando

[2] `https://web.getmonero.org/pt-br/`
[3] `https://bytecoin.org/`
[4] `https://electroneum.com/`

sua detecção por mecanismos tradicionais baseados em assinaturas. A biblioteca Coinhive, pioneira nesse movimento, utilizou *wasm* para incrementar a taxa de *hashes* por segundo em ataques de *drive-by mining*, consolidando-se como uma das principais ferramentas em campanhas massivas de mineração não autorizada (Konoth et al., 2018; Harnes and Morrison, 2024b).

Diante da baixa eficácia de mecanismos tradicionais de detecção, pesquisas recentes têm explorado abordagens mais robustas, como análise dinâmica, aprendizado de máquina e extração de padrões de fluxo. Técnicas baseadas em ICFG (*Inter-Procedural Control Flow Graph*), métricas de entropia e inspeção de memória em tempo de execução têm demonstrado eficácia na detecção de módulos *wasm* maliciosos, mesmo na presença de técnicas de ofuscação, fragmentação de execução ou evasão adaptativa (Harnes and Morrison, 2024b; Kaza et al., 2024).

Adicionalmente, a estrutura modular do *WebAssembly* permite que agentes maliciosos embaralhem o fluxo de controle por meio de instruções indiretas (`call_indirect`) e saltos condicionais (`br_if`), dificultando a construção precisa de CFG (Harnes and Morrison, 2024b). A comunicação entre os módulos *wasm* e os servidores de mineração é frequentemente estabelecida via *WebSocket*, o que dificulta a análise de tráfego e reduz a eficácia de técnicas de inspeção profunda de pacotes (Konoth et al., 2018).

Essas capacidades tornam o *WebAssembly* uma ferramenta poderosa não apenas para aplicações legítimas, mas também para agentes maliciosos interessados em explorar o navegador como vetor de ataque. Nesse contexto, destaca-se o fenômeno da mineração não autorizada, tema da próxima subseção.

2.2.2 Adoção do WebAssembly em Campanhas Maliciosas

Com o avanço das tecnologias de execução no lado do cliente, a mineração não autorizada consolidou-se como uma das principais ameaças silenciosas no ecossistema web. Diferente de abordagens ruidosas como o *ransomware*, essa prática visa explorar os recursos computacionais da vítima de forma contínua e furtiva, comprometendo o desempenho dos dispositivos e degradando a experiência de navegação sem qualquer interação ou percepção explícita do usuário. Essa transição estratégica tem levado agentes maliciosos a priorizarem métodos de extração de valor menos intrusivos e mais sustentáveis, como a mineração não autorizada de cripto moedas. Para maximizar o tempo de atividade e reduzir o risco de detecção, os operadores dessas campanhas adotam abordagens adaptativas, variando constantemente suas ferramentas, táticas e procedimentos (Crypto Potato, 2023).

O *cryptojacking* refere-se ao uso não autorizado dos recursos computacionais de um dispositivo para a mineração de cripto moedas, geralmente realizado de forma silenciosa durante a navegação em páginas web. Nesse tipo de ataque, scripts executáveis frequentemente implementados em JavaScript ou *WebAssembly* são incorporados a páginas web comprometidas ou maliciosas, sendo automaticamente executados no navegador do usuário com o objetivo de realizar operações de mineração sem o seu consentimento (Konoth et al., 2018; Musch et al., 2019; Harnes and Morrison, 2024b).

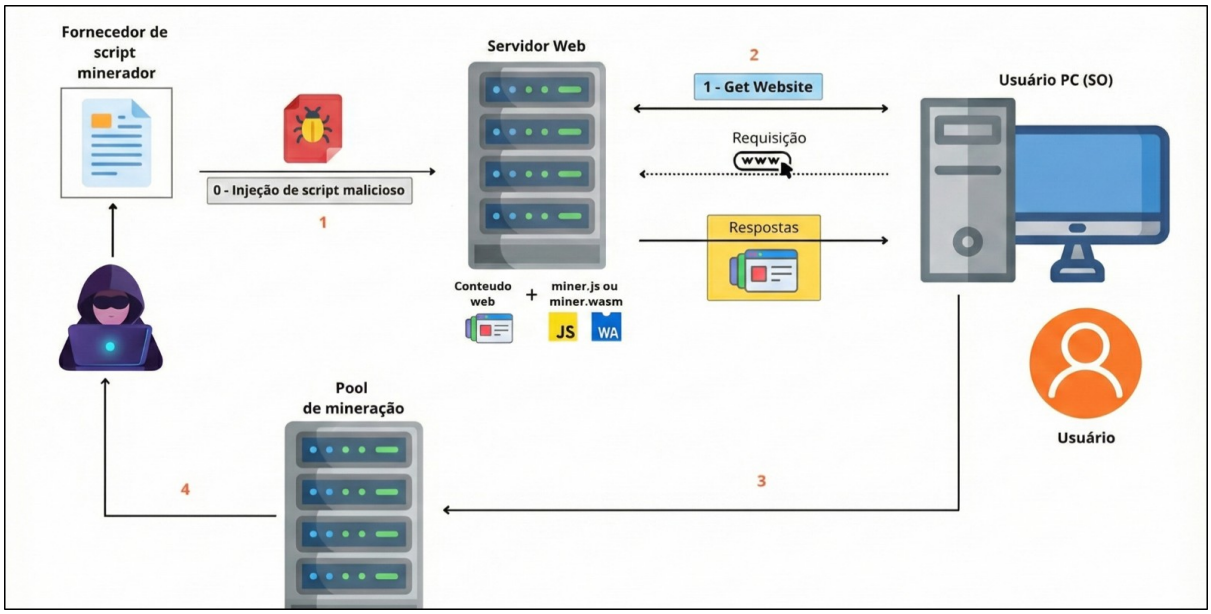
Esse modelo de ataque ganhou notoriedade a partir de 2017, período marcado pela rápida valorização de cripto moedas e pelo surgimento de serviços de mineração em navegador. Entre esses serviços, destaca-se a plataforma Coinhive, que permitia a mineração da criptomoeda Monero diretamente em navegadores web. Embora inicialmente proposta como alternativa à monetização baseada em publicidade, a plataforma passou a ser amplamente explorada em campanhas de mineração não autorizada, contribuindo para a disseminação do *cryptojacking* em larga escala (Konoth et al., 2018).

Ao contrário de ataques tradicionais baseados em *malware* instalável, o *cryptojacking* opera inteiramente no ambiente do navegador, dispensando a instalação de software no dispositivo da vítima. Essa característica torna sua detecção mais complexa, uma vez que o código de mineração é carregado dinamicamente durante a navegação e pode ser encerrado automaticamente quando o usuário deixa a página. Além disso, muitos mineradores implementam técnicas de evasão, como limitação do uso de CPU,

fragmentação de execução e ofuscação de código, com o objetivo de reduzir a probabilidade de detecção (Romano et al., 2020; Naseem et al., 2021).

Atualmente, o Bitcoin é minerado apenas com hardware específico como ASICs para mineração eficaz, o que faz com que as moedas alternativas sejam as mais mineradas pelos *malwares*, uma vez que essas não precisam de hardware especializado. O aumento no valor de mercado das criptomoedas levou ao desenvolvimento de serviços de mineração de criptomoedas, como o Coinhive, que pode ser facilmente integrado a sites para monetizar o poder computacional de seus visitantes (Konoth et al., 2018).

Figura 2.2: Fluxo de ataque de *cryptojacking*.



Do ponto de vista técnico, campanhas modernas de *cryptojacking* frequentemente combinam código JavaScript responsável pela orquestração da execução com módulos *WebAssembly* responsáveis pela implementação das rotinas de mineração. Essa arquitetura híbrida permite que operações computacionalmente intensivas, como funções de *hash* utilizadas em algoritmos de prova de trabalho, sejam executadas com maior eficiência dentro do navegador. Como consequência, a identificação desses módulos tornou-se um problema relevante para pesquisas voltadas à detecção de atividades maliciosas baseadas em *WebAssembly*.

A Figura 2.2 ilustra o fluxo de um ataque de *cryptojacking*, o qual, do ponto de vista operacional, apresenta similaridades com o processo de criptomineração legítima. A seguir, descreve-se detalhadamente cada etapa desse tipo de ataque:

1. **Comprometimento de um site legítimo**

   - Um agente de ameaça (*Threat Actor*) compromete um site legítimo, explorando vulnerabilidades para injetar um script de mineração baseado em JavaScript, *WebAssembly* ou outra tecnologia executável no navegador. Esse código malicioso é configurado para ser automaticamente ativado sempre que a página comprometida é acessada por um usuário.

2. **Execução do script no navegador da vítima**

   - Os usuários acessam o site comprometido sem perceber que este contém um script malicioso embutido em seu código.
   - Ao ser carregada no navegador, a página ativa automaticamente o script, que passa a utilizar os recursos computacionais do dispositivo da vítima para realizar a mineração de criptomoedas.

Esse processo ocorre de forma silenciosa, sem requerer qualquer interação direta do usuário e, frequentemente, sem detecção imediata.

3. **Uso não autorizado dos recursos computacionais da vítima**
   - O código malicioso executa algoritmos de mineração diretamente no dispositivo da vítima, consumindo intensivamente ciclos de CPU e/ou GPU.
   - Esse uso não autorizado pode resultar em degradação do desempenho do sistema, aquecimento excessivo do hardware e aumento significativo no consumo de energia, afetando negativamente a experiência de uso. Em ambientes corporativos, tais impactos podem comprometer a eficiência operacional e a disponibilidade de recursos computacionais críticos.
4. **Transferência dos ganhos para o agente mal-intencionado**
   - A criptomoeda obtida por meio do processo de mineração é automaticamente transferida para a carteira digital controlada pelo agente de ameaça, assegurando-lhe ganhos financeiros sem a necessidade de investimentos em infraestrutura própria.

Esse método de ataque se destaca por sua dificuldade de detecção, uma vez que não exige a instalação de *malware* tradicional e opera inteiramente no ambiente do navegador. Sua natureza furtiva, combinada com a simplicidade de implantação e o retorno financeiro direto, torna o *cryptojacking* uma ameaça persistente, demandando mecanismos específicos para a detecção de comportamentos anômalos em tempo de execução.

2.3 Análise de Binários *WebAssembly*

A análise de binários *WebAssembly* tem se tornado um tema relevante no contexto da segurança de aplicações web, em razão da crescente adoção dessa tecnologia para execução de código de alto desempenho em navegadores e ambientes de execução distribuídos. Diferentemente do código JavaScript tradicional, os módulos *WebAssembly* são distribuídos em formato binário compilado, o que dificulta sua inspeção direta e limita a eficácia de técnicas de detecção baseadas em análise textual ou assinaturas simples (Lonkar and Chandrayan, 2018).

Nesse contexto, diversos estudos têm investigado métodos para analisar e compreender o comportamento de binários *WebAssembly*. O estudo conduzido por Harnes and Morrison (2024b) apresenta uma análise sistemática das principais técnicas e ferramentas empregadas para esse propósito, considerando diferentes cenários de aplicação, incluindo segurança de aplicações web, computação em nuvem e computação de borda. Os autores propõem uma categorização das abordagens de análise baseada em seus objetivos principais, identificando três classes predominantes:

- (i) detecção de binários *WebAssembly* maliciosos;
- (ii) identificação de vulnerabilidades em módulos *WebAssembly*;
- (iii) análise de contratos inteligentes implementados nessa tecnologia.

De forma geral, as técnicas utilizadas para análise de binários *WebAssembly* podem ser agrupadas em três categorias principais: análise estática, análise dinâmica e análise híbrida. Cada uma dessas abordagens apresenta diferentes vantagens e limitações em termos de precisão, custo computacional e capacidade de lidar com técnicas de evasão utilizadas por códigos maliciosos.

A análise de binários *WebAssembly* frequentemente requer a transformação do código binário em representações estruturais que permitam examinar sua organização sintática e semântica. Diferentemente

de linguagens interpretadas, os módulos *WebAssembly* são distribuídos em formato binário compacto e estruturado, o que dificulta sua inspeção direta e limita a aplicação de técnicas tradicionais de análise baseadas em código textual ou em assinaturas simples (Haas et al., 2017; Lonkar and Chandrayan, 2018).

As subseções a seguir descrevem essas categorias de análise com ênfase em sua aplicação na detecção de códigos maliciosos baseados em *WebAssembly*, incluindo campanhas de *cryptojacking* e outras formas de abuso de recursos computacionais em navegadores.

Análise Estática

A análise estática examina o código-fonte ou o binário de um programa sem a necessidade de executá-lo. Essa abordagem é geralmente rápida e de baixo custo computacional. No contexto da detecção de *cryptojacking*, técnicas de análise estática podem ser aplicadas de diferentes formas (Harnes and Morrison, 2024b):

- **Análise de Assinaturas e Padrões:** Uma das estratégias mais diretas consiste na identificação de características inerentes ao código de mineração. Isso inclui a identificação de funções criptográficas específicas ou de "impressões digitais"(*fingerprints*) construídas com base na frequência de instruções como `XOR` e rotações de bits, frequentemente utilizadas em algoritmos como o *CryptoNight* (Kim et al., 2022).

- **Análise de Grafo de Fluxo de Controle (CFG):** Métodos mais avançados reconstroem a estrutura do programa através de um Grafo de Fluxo de Controle. A análise desses grafos possibilita a identificação de padrões de execução, como laços computacionalmente intensivos, característicos de algoritmos de *hash* empregados em mineração. Algumas técnicas conseguem, inclusive, analisar código JavaScript e asm.js, convertendo-os primeiramente para *webassembly* para criar um grafo unificado do comportamento do programa (Kim et al., 2022; Harnes and Morrison, 2024b).

- **Classificação baseada em Aprendizado de Máquina**: Uma abordagem emergente converte o binário *webassembly* em uma representação visual, como uma imagem em escala de cinza, e utiliza uma CNN *(Convolutional Neural Network)* para classificar o arquivo como benigno ou malicioso. Essa técnica apresenta elevada eficiência computacional, sendo particularmente adequada para sistemas de detecção em tempo real (Harnes and Morrison, 2024b).

Apesar da eficiência e baixo custo computacional, a principal limitação das abordagens estáticas reside em sua suscetibilidade a técnicas de ofuscação de código, que alteram a estrutura superficial dos binários com o intuito de evadir mecanismos de detecção baseados em assinaturas ou padrões reconhecíveis (Eskandari et al., 2018; Harnes and Morrison, 2024b; Kim et al., 2022).

Análise Dinâmica

A análise dinâmica consiste no monitoramento do comportamento de um programa durante sua execução, com o objetivo de identificar atividades potencialmente maliciosas. Essa abordagem é, por natureza, mais resiliente à ofuscação do código, embora costume implicar em maior sobrecarga de desempenho. As principais técnicas dinâmicas incluem (Harnes and Morrison, 2024b):

- **Monitoramento de Recursos do Sistema:** A detecção pode ser realizada pela observação de padrões anormais no uso de recursos computacionais, como consumo elevado e contínuo de CPU, uso intensivo de memória e aumento no tráfego de rede. Esses sinais são frequentemente associados à presença de atividades de mineração de criptomoedas (Kim et al., 2022; Harnes and Morrison, 2024b).

- **Análise Comportamental com Aprendizado de Máquina:** Diversas soluções realizam a coleta de características comportamentais durante a execução, tais como o número de *Web Workers* instanciados, o uso de *WebSockets* para comunicação com *pools* de mineração e chamadas a APIs específicas do JavaScript. Esses dados são então analisados por classificadores baseados em aprendizado de máquina, como SVM, com o objetivo de detectar comportamentos associados à atividade maliciosa (Harnes and Morrison, 2024b).

- **Análise de Fluxo de Instruções:** Esta técnica consiste em examinar a sequência ou a frequência das instruções executadas durante a execução do programa. Aplicações de mineração apresentam comportamento repetitivo e uniforme, realizando grandes volumes de cálculos de *hash*, o que resulta em padrões de instruções distintivos quando comparados a aplicações benignas (Kim et al., 2022; Harnes and Morrison, 2024b).

- **Instrumentação e Monitoramento em Linha (IRM):** Abordagens mais sofisticadas instrumentam o binário *WebAssembly* por meio da inserção de contadores ou trechos de código de monitoramento, com o objetivo de extrair características semânticas durante a execução. Essas informações são então comparadas a assinaturas previamente conhecidas de criptomineradores. Embora apresente alta precisão, essa técnica tende a incorrer em elevado *overhead* de desempenho (Harnes and Morrison, 2024b).

A análise dinâmica tem se mostrado uma alternativa eficaz para a detecção de criptomineradores que empregam técnicas de evasão, especialmente aquelas baseadas em ofuscação e diversificação de binários *WebAssembly*. Ao observar diretamente o comportamento de execução dos programas, essas abordagens são capazes de identificar padrões maliciosos que não seriam evidentes em análises estáticas. No entanto, sua adoção em larga escala ainda é limitada pelas exigências computacionais envolvidas, pela necessidade de ambientes controlados e pela complexidade de instrumentação. Assim, embora robusta, a análise dinâmica exige um equilíbrio cuidadoso entre profundidade analítica e viabilidade operacional, sendo mais apropriada para contextos onde a acurácia na detecção justifica o custo de monitoramento em tempo real (Harnes and Morrison, 2024b; Kim et al., 2022).

Análise Híbrida

A análise híbrida combina técnicas estáticas e dinâmicas com o objetivo de reunir as vantagens de ambas: a eficiência e rapidez da análise estática, aliadas à robustez e profundidade da análise dinâmica. Essa abordagem é considerada promissora no contexto da detecção de *cryptojacking*, embora ainda seja pouco explorada na literatura especializada (Kim et al., 2022; Harnes and Morrison, 2024b). A seguir, são apresentados alguns exemplos de implementações híbridas:

- Análise Estática para Filtragem: Consiste em aplicar a análise estática como etapa inicial para identificar rapidamente funções ou binários potencialmente suspeitos. Esses elementos são, em seguida, encaminhados para uma análise dinâmica mais aprofundada, com o objetivo de confirmar comportamentos maliciosos e reduzir a ocorrência de falsos positivos (Harnes and Morrison, 2024b).

- Combinação de Fontes de Dados: Integra a análise estática do código binário com o monitoramento dinâmico de recursos em tempo de execução. Por exemplo, uma técnica pode detectar estaticamente um padrão de mineração suspeito e, simultaneamente, observar padrões de acesso ao cache da CPU durante a execução, a fim de confirmar a atividade maliciosa (Harnes and Morrison, 2024b).

- Verificação de Consentimento: Uma abordagem relevante combina a análise estática do código, visando identificar algoritmos de mineração, com a análise dinâmica da interface do usuário, a

fim de verificar se houve consentimento explícito por parte do usuário. Essa combinação permite distinguir entre *cryptojacking* e mineração legítima (Harnes and Morrison, 2024b).

Além disso, aspectos como a escalabilidade, o consumo de recursos computacionais e a complexidade de implementação ainda limitam sua aplicabilidade em cenários de larga escala ou em ambientes restritos, como dispositivos embarcados. Apesar dessas limitações, a análise híbrida tem se destacado como um caminho viável para aumentar a precisão na detecção de *cryptojacking*, especialmente diante de técnicas sofisticadas de ofuscação e evasão que comprometem abordagens unilaterais (Harnes and Morrison, 2024b; Kim et al., 2022).

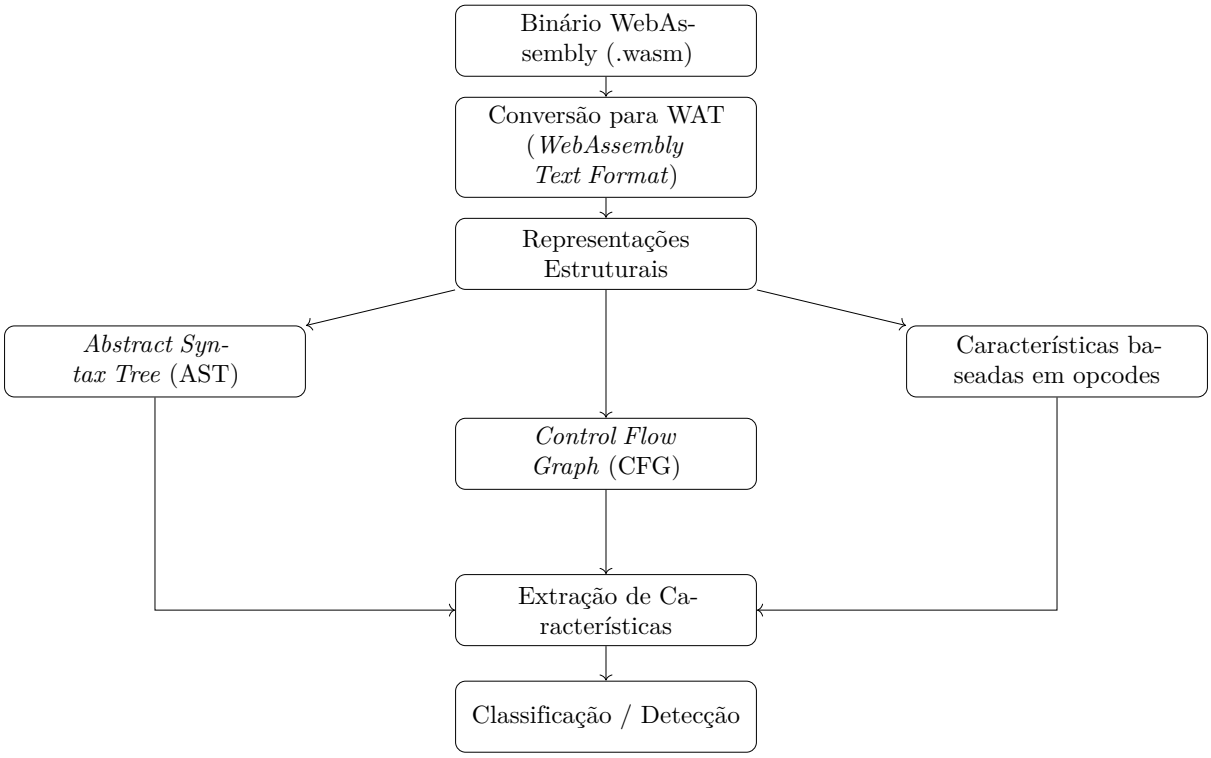
À medida que técnicas de análise estrutural de binários *WebAssembly* evoluem, observa-se um crescente interesse na utilização de métodos automatizados capazes de interpretar padrões complexos presentes no código. Representações como AST, CFG e distribuições de *opcodes* permitem descrever propriedades estruturais e comportamentais dos programas, servindo como base para a extração de características utilizadas em tarefas de classificação automática. Entretanto, abordagens tradicionais de aprendizado de máquina frequentemente dependem de conjuntos de atributos previamente definidos, o que pode limitar sua capacidade de capturar relações semânticas mais profundas presentes no código.

2.3.1 Representação Estrutural de Binários *WebAssembly*

Para mitigar a fragilidade da análise estática perante a ofuscação, sem incorrer nos altos custos de desempenho da análise dinâmica, diversos trabalhos na literatura têm investigado métodos de engenharia reversa capazes de converter módulos *WebAssembly* em representações intermediárias mais adequadas à extração de características.

Uma etapa frequentemente empregada nesse processo consiste na conversão do módulo binário para o formato textual denominado `.wat`. Ferramentas como o WABT (*WebAssembly Binary Toolkit*) permitem realizar essa transformação preservando a estrutura interna do módulo, incluindo as suas seções de tipos, funções, memória e exportações. Essa representação textual facilita a inspeção manual e automatizada do código, permitindo observar explicitamente as instruções executadas e a organização das estruturas de controle (Rossberg et al., 2024).

Figura 2.3: Fluxo de transformação de binários WebAssembly em representações estruturais utilizadas para extração de características e classificação automática de código.



A partir dessa representação textual, diferentes estruturas analíticas podem ser derivadas. Em vez de analisar o código como uma simples sequência linear de *bytes*, o binário é transformado em grafos e árvores que capturam o fluxo de dados e de controlo subjacente. As duas abstrações mais proeminentes na análise de *malware* são a AST e o CFG:

- A Árvore de Sintaxe Abstrata é uma representação em formato de árvore da estrutura sintática do código fonte. Como o *WebAssembly* funciona através de uma máquina baseada em pilha, o seu *bytecode* ou representação `.wat` pode ser facilmente convertido numa AST onde os nós folha representam operandos e os nós interiores representam as instruções que operam sobre eles. Esta representação é imune a mudanças superficiais de sintaxe e preserva o contexto hierárquico das funções matemáticas exigidas por algoritmos de mineração.

- O Grafo de Fluxo de Controlo representa todos os caminhos que podem ser atravessados por um programa durante a sua execução. Num CFG de um módulo *wasm*, os nós correspondem a blocos básicos (*basic blocks*) de instruções que executam sequencialmente, e as arestas dirigidas representam os saltos de controlo de fluxo (como as instruções `br` e `br_if`). A extração de CFGs é vital para identificar *malware*, pois algoritmos complexos, como os utilizados no *cryptojacking*, possuem assinaturas estruturais em grafo que são quase impossíveis de mascarar completamente através da ofuscação simples (Guo et al., 2020).

Estudos recentes sobre detecção de *cryptojacking* exploram diferentes formas de representação estrutural e semântica do código. O sistema MineSweeper, por exemplo, propõe a identificação de códigos de mineração a partir de características intrínsecas de funções criptográficas, incluindo padrões recorrentes de operações aritméticas e bit a bit associadas ao cálculo de funções de *hash* (Konoth et al., 2018). De forma semelhante, o sistema Outguard utiliza um navegador instrumentado para coletar características relacionadas ao comportamento de execução de páginas web, incluindo informações sobre o uso de *Web Workers*,

chamadas de rede e presença de módulos *WebAssembly*, empregando posteriormente um classificador baseado em SVM para distinguir páginas benignas de páginas que executam mineração não autorizada (Kharraz et al., 2019). Os resultados experimentais indicam que a abordagem baseada em aprendizado de máquina alcança elevadas taxas de detecção ao considerar múltiplas fontes de características.

Outra linha de pesquisa busca explorar características semânticas diretamente associadas ao comportamento de mineração. Nesse contexto, o sistema MinerRay propõe um método de análise que infere propriedades essenciais dos algoritmos a partir da observação de padrões computacionais relacionados ao cálculo repetitivo de funções de *hash*, tanto em código JavaScript quanto em módulos *WebAssembly* (Romano et al., 2020). Essa abordagem evita depender exclusivamente de assinaturas específicas ou implementações particulares de algoritmos criptográficos.

Além dessas representações estruturais, diversas abordagens utilizam características derivadas diretamente do conjunto de instruções executadas pelos módulos *WebAssembly*. Entre essas características destacam-se histogramas de *opcodes*, frequências relativas de operações aritméticas e bit a bit, bem como padrões de acesso à memória linear.

No contexto da segurança de sistemas, a utilização de representações estruturais como AST e CFG, combinadas com métodos de extração de características baseados em instruções, permite construir modelos de classificação capazes de distinguir módulos benignos de módulos associados a comportamentos maliciosos. Contudo, a construção destas representações perfeitas a partir do *WebAssembly* apresenta desafios. A presença de chamadas indiretas a funções através de tabelas (instrução `call_indirect`) introduz ambiguidades na resolução dos caminhos do grafo durante uma análise estática (Harnes and Morrison, 2024b).

Apesar de estas etapas introduzirem custos adicionais de processamento, especialmente quando aplicadas à análise de grandes volumes de módulos coletados na web, a literatura recente indica que as representações estruturais e semânticas do código constituem a base metodológica mais robusta para o desenvolvimento de sistemas de detecção automatizada de *malware* e de *cryptojacking*.

2.3.2 Extração de Features

A eficácia de qualquer sistema de detecção de comportamento malicioso baseado em aprendizagem automática depende de forma crítica da qualidade dos atributos extraídos dos dados brutos. No domínio da análise de *malware*, este processo é formalmente conhecido como Extração de Características (*Feature Extraction*), e atua como a ponte fundamental entre o código analisado e os algoritmos de classificação.

Matematicamente, uma função de extração de características (ou função de *features*) pode ser definida como um mapeamento $\phi : \mathcal{X} \rightarrow \mathcal{F}$, onde $\mathcal{X}$ representa o espaço de entrada (os módulos *WebAssembly* brutos ou as suas representações estruturais, como AST e CFG) e $\mathcal{F}$ representa o espaço de características (*feature space*), tipicamente um vetor numérico em $\mathbb{R}^n$. O objetivo de ϕ é isolar as propriedades intrínsecas e discriminativas de $x \in \mathcal{X}$ que permitam separar amostras benignas de amostras maliciosas.

No contexto específico do *WebAssembly*, as características extraídas enquadram-se habitualmente em três taxonomias principais (Romano et al., 2020; Kharraz et al., 2019):

- **Características Sintáticas:** Extraídas diretamente do *bytecode*. Incluem a frequência de *opcodes* histogramas de instruções, n-gramas de *bytes*, proporção de operações aritméticas ou de manipulação de memória (cruciais para detetar algoritmos de mineração baseados em *hashes*) e a listagem de chamadas a APIs importadas.

- **Características Estruturais:** Derivadas das representações em grafo AST e CFG. Estas métricas capturam a complexidade ciclomática do módulo `.wasm`, a profundidade máxima da árvore de

sintaxe, o número de caminhos de execução independentes e os padrões de saltos de fluxo (`br`, `br_if`), que tendem a revelar a arquitetura de *loops* contínuos típicos no *cryptojacking*.

- **Características Semânticas e Dinâmicas:** Relacionadas com o comportamento de execução do módulo no ambiente *web*, tais como taxas de acesso à memória linear, instanciação de *Web Workers* ou interações com a rede através de *WebSockets* para a submissão de *proof-of-work* a *pools* de mineração.

Tradicionalmente, a engenharia de características (*Feature Engineering*) dependia de analistas humanos ou heurísticas estatísticas para conceber manualmente a função ϕ. Contudo, com a proliferação de técnicas de evasão e de ofuscação binária no *WebAssembly* (Cabrera-Arteaga et al., 2023), as características mapeadas manualmente tornaram-se insuficientes. Essa limitação impulsionou a adoção de métodos de aprendizagem de representações (*Representation Learning*), nos quais modelos avançados de IA assumem a responsabilidade de aprender e otimizar a função de *features* de forma automatizada, gerando vetores densos altamente dimensionais.

2.3.3 Heurísticas e Padrões Sintáticos Associados à Mineração

A detecção de *cryptojacking* em *WebAssembly* fundamenta-se, em parte, na identificação de padrões sintáticos característicos das rotinas de mineração de criptomoedas. Algoritmos de prova de trabalho (*Proof-of-Work* - PoW), como *CryptoNight* e *RandomX*, amplamente utilizados por criptomineradores, impõem exigências computacionais específicas que se manifestam no código como assinaturas sintáticas observáveis (Loose et al., 2023; Cabrera-Arteaga et al., 2023).

Características dos Algoritmos de Mineração

Os algoritmos PoW empregados por criptomoedas como Monero (XMR) são projetados para serem intensivos em memória e em operações aritméticas, dificultando a implementação eficiente em hardware especializado ASICs (Loose et al., 2023). No contexto de *WebAssembly*, essas características traduzem-se em padrões recorrentes:

- **Operações aritméticas de 64 bits**: a manipulação de inteiros de 64 bits (`I64.add`, `I64.xor`, `I64.shl`) é predominante em algoritmos como *RandomX*, que opera sobre palavras de 64 bits (Cabrera-Arteaga et al., 2023);
- **Acessos intensivos à memória linear**: a geração de hashes requer leituras e escritas frequentes em grandes buffers de memória, refletindo-se em instruções de carregamento e armazenamento (`I64.load`, `I64.store`) em alta densidade;
- **Estruturas de repetição**: laços (*loops*) são utilizados para iterar sobre os blocos de dados durante o processo de hashing, resultando em estruturas de controle de fluxo repetitivas (Loose et al., 2023);
- **Operações bit a bit**: funções de hash envolvem operações como `I64.and`, `I64.or` e `I64.xor` para manipulação de bits, cuja frequência é anormalmente elevada em mineradores (Cabrera-Arteaga et al., 2023).

Heurísticas Determinísticas para Detecção

Com base nessas características, a literatura tem explorado heurísticas determinísticas extraídas da AST como indicadores de atividade de mineração. Estudos demonstram que a frequência de operações aritméticas de 64 bits, a densidade de acessos à memória linear e a presença de estruturas de repetição

aninhadas correlacionam-se fortemente com a presença de rotinas de *hashing* (Loose et al., 2023; Cabrera-Arteaga et al., 2023).

Formalmente, podem ser definidas as seguintes características sintáticas mensuráveis:

$$\mathbf{f} = [\text{count}(I64_{ops}), \text{count}(I64_{load}), \text{count}(I64_{store}), \text{count}(Loop), \text{count}(I64_{bitwise})] \tag{2.1}$$

Onde:

- count($I64_{ops}$): número total de instruções aritméticas de 64 bits (add, sub, mul, etc.);
- count($I64_{load}$) e count($I64_{store}$): contagem de acessos à memória linear;
- count($Loop$): número de estruturas de repetição no módulo;
- count($I64_{bitwise}$): número de operações bit a bit de 64 bits.

A combinação dessas heurísticas proporciona indicadores sintáticos robustos para diferenciar aplicações benignas de scripts de mineração não autorizada. Embora não sejam suficientes isoladamente dada a possibilidade de ofuscação, quando associadas a representações semânticas em abordagens híbridas, contribuem para a construção de classificadores mais resilientes a ataques adversariais (Ahmadi, 2025; Loose et al., 2023).

Relevância para a Arquitetura Proposta

No contexto desta pesquisa, as heurísticas descritas são extraídas por meio de análise estática da AST dos módulos *WebAssembly*, gerando um vetor de características determinísticas $\mathbf{t} \in \mathbb{R}^{D_t}$. Este vetor é concatenado às representações semânticas $\mathbf{s}$ geradas por modelos de linguagem, formando o espaço vetorial apresentado na Seção 2.5. A fusão de evidências sintáticas e semânticas visa aumentar a eficiência do classificador diante de técnicas de ofuscação que poderiam enganar abordagens baseadas exclusivamente em uma das fontes.

2.4 Machine Learning

Formalmente, o Aprendizado de Máquina - ML (*Machine Learning*) é definido como a arte e a ciência de programar computadores para que estes extraiam conhecimento a partir de dados, permitindo que os sistemas melhorem iterativamente o seu desempenho numa tarefa específica sem a necessidade de serem explicitamente programados com regras estáticas (Géron, 2022).

Os algoritmos de aprendizado de máquina são tradicionalmente categorizados em três paradigmas principais (Géron, 2022):

- **Aprendizado Supervisionado:** baseia-se na utilização de conjuntos de dados previamente rotulados para inferir uma função de mapeamento entre as variáveis de entrada e as saídas esperadas. É predominantemente aplicado em tarefas de classificação e regressão.
- **Aprendizado Não Supervisionado:** atua sobre dados sem anotação prévia, com o intuito de identificar padrões subjacentes e modelar a estrutura latente inerente à distribuição da informação, sem depender de variáveis-alvo predefinidas.
- **Aprendizado por Reforço:** fundamenta-se na interação de um agente computacional com um ambiente dinâmico, onde a otimização da política de decisão ocorre por meio de um sistema contínuo de recompensas e penalidades.

No contexto da segurança da informação, o aprendizado de máquina tem se consolidado como uma abordagem fundamental para a detecção de *malware*, permitindo a análise de grandes volumes de dados e a identificação de padrões complexos que métodos tradicionais baseados em assinaturas não conseguem capturar (Berrios et al., 2025; Ahmadi, 2025). Diferentemente das abordagens estáticas que falham diante de técnicas de ofuscação ou polimorfismo os modelos de aprendizado de máquina podem generalizar padrões estruturais e comportamentais subjacentes, possibilitando a detecção automatizada de ameaças conhecidas e *zero-day* (Manthena et al., 2025b).

Esta pesquisa enquadra-se no paradigma supervisionado, abordando a tarefa de classificação binária de binários *wasm* como benignos ou maliciosos. A escolha justifica-se pela necessidade de distinguir, com elevada precisão, amostras de código legítimo daquelas que implementam atividades de mineração de criptomoedas não autorizadas (Loose et al., 2023).

2.4.1 Aprendizado Supervisionado

Conforme apresentado na seção anterior, o aprendizado supervisionado constitui o paradigma central desta pesquisa. Neste contexto, o modelo é treinado a partir de um conjunto de dados rotulados, aprendendo uma função $f : \mathcal{X} \to \mathcal{Y}$ que mapeia características extraídas (vetores de atributos) a classes discretas (Géron, 2022). Para problemas de classificação binária como o que se apresenta na detecção de binários Wasm maliciosos, o objetivo é estimar a probabilidade de uma amostra pertencer à classe positiva *malware* ou negativa (benigno), com base em padrões estruturais e comportamentais aprendidos durante a fase de treinamento (Bishop, 2006). A escolha do algoritmo adequado influencia diretamente a capacidade de generalização do modelo, especialmente diante de técnicas de ofuscação e polimorfismo comuns em ameaças como o *cryptojacking* (Manthena et al., 2025a).

2.4.2 Algoritmos de Classificação

Nesta seção, descrevem-se quatro algoritmos de classificação supervisionada amplamente consolidados na literatura de detecção de *malware* e análise de segurança: Regressão Logística, *Random Forest*, *XGBoost* e *SVM* com *kernel* linear. Cada qual possui fundamentos teóricos distintos, sendo selecionados para compor o arcabouço metodológico da presente pesquisa.

Regressão Logística

A Regressão Logística é um modelo estatístico para classificação binária que estima a probabilidade de uma instância pertencer a uma determinada classe por meio da função sigmoide (Hosmer Jr et al., 2013). Formalmente, o modelo expressa a probabilidade

$$P(Y = 1 \mid \mathbf{x}) = \frac{1}{1 + e^{-(\beta_0 + \boldsymbol{\beta}^T \mathbf{x})}}, \tag{2.2}$$

onde $\mathbf{x}$ representa o vetor de características extraídas do binário *wasm*. Diferentemente da regressão linear, a Regressão Logística não exige normalidade dos resíduos e seus coeficientes β_j são interpretados como log-razão de chances (*log-odds*), o que a torna útil para análise de importância de atributos James et al. (2021). É considerada um modelo linear de decisão, sendo computacionalmente eficiente e adequada como *baseline* em tarefas de detecção de *malware* Ahmadi (2025). Sua principal limitação reside na incapacidade de capturar relações não lineares entre as variáveis preditoras e a variável resposta sem transformações explícitas ou engenharia de atributos.

Random Forest

Random Forest é um método de *ensemble* proposto por Breiman (2001), que constrói múltiplas árvores de decisão a partir de amostras *bootstrap* dos dados e seleciona aleatoriamente um subconjunto de variáveis preditoras em cada nó. A combinação das árvores por votação majoritária reduz a variância do modelo sem aumentar excessivamente o viés, tornando o algoritmo robusto a *overfitting* e capaz de lidar com dados de alta dimensionalidade uma característica particularmente relevante na análise de binários *wasm*, nos quais centenas ou milhares de atributos estruturais podem ser extraídos (Hastie et al., 2009). Conforme Liaw and Wiener (2002), uma vantagem adicional da Random Forest é a possibilidade de estimar a importância de cada variável preditora (*feature importance*), o que auxilia na interpretação do modelo e na redução de dimensionalidade. Na detecção de *cryptojacking*, a Random Forest tem demonstrado eficácia na identificação de padrões ofuscados que métodos baseados em assinaturas não detectam (Loose et al., 2023). Como desvantagem, o algoritmo é menos interpretável que modelos lineares e pode exigir maior capacidade computacional em conjuntos de dados muito grandes.

XGBoost

O XGBoost (eXtreme Gradient Boosting), proposto por Chen and Guestrin (2016), é uma implementação eficiente e escalável do *Gradient Boosting Machine* (Friedman, 2001). O algoritmo constrói árvores sequencialmente, onde cada nova árvore corrige os resíduos da anterior, atribuindo maiores pesos às instâncias mal classificadas. Diferentemente do *bagging* utilizado na (*Random Forest*), o *boosting* reduz predominantemente o viés, embora possa ser suscetível a *overfitting* se não regularizado adequadamente (Nielsen, 2016). O XGBoost destaca-se por incorporar regularização L_1 (LASSO) e L_2 (Ridge), poda de árvores (*pruning*), paralelização e manipulação eficiente de valores faltantes características que o tornam particularmente adequado para conjuntos de dados desbalanceados, uma situação comum em detecção de *malware* (Berrios et al., 2025). Devido ao seu alto desempenho preditivo, o XGBoost tem sido amplamente adotado em competições de ciência de dados e em aplicações de segurança, incluindo a detecção de *cryptojacking* em *WebAssembly*.

Support Vector Machine (SVM)

A Support Vector Machine, formulada por Cortes and Vapnik (1995), é um classificador que busca o hiperplano de separação que maximiza a margem entre as classes. Na versão linear, assume-se que os dados são linearmente separáveis no espaço original de características, e o hiperplano ótimo é encontrado resolvendo-se um problema de otimização quadrática, sendo os vetores de suporte os pontos críticos mais próximos da fronteira de decisão (Bishop, 2006). A SVM linear apresenta vantagens em termos de interpretabilidade e custo computacional, especialmente em problemas de alta dimensionalidade, como a classificação de binários *wasm* com centenas de atributos extraídos via análise estática. No entanto, exige normalização prévia dos dados (escalonamento) e não captura relações não lineares a menos que se utilize o *kernel trick* (Schölkopf and Smola, 2002). Para os propósitos deste trabalho, adota-se o *kernel* linear por sua simplicidade, eficiência computacional e desempenho competitivo em tarefas de classificação de *malware* quando as características são bem estruturadas (Manthena et al., 2025a).

A escolha dos algoritmos fundamenta-se na diversidade de suas abordagens teóricas e na frequência de sua aplicação em estudos sobre detecção de *malware* em *WebAssembly*. A Regressão Logística é utilizada como modelo de referência (*baseline*) por ser um classificador linear probabilístico de menor complexidade estrutural, o que permite estabelecer um patamar mínimo de desempenho para a avaliação comparativa dos demais métodos (James et al., 2021). O *Random Forest* e o XGBoost representam as abordagens de *bagging* e *boosting*, respectivamente, sendo empregados pela capacidade de processar atributos estruturais

e comportamentais com menor susceptibilidade ao sobreajuste (*overfitting*) (Hastie et al., 2009; Chen and Guestrin, 2016). Por fim, o SVM com núcleo linear baseia-se na teoria da margem máxima, sendo adequado para o tratamento de dados em espaços de alta dimensionalidade, característicos da análise estática de binários (Cortes and Vapnik, 1995).

A Tabela 2.1 sintetiza as principais características comparativas entre os quatro algoritmos no contexto da detecção de ameaças. Cabe ressaltar que não existe um algoritmo universalmente superior; o desempenho relativo depende intrinsecamente das características específicas do conjunto de dados incluindo o balanceamento entre classes, a qualidade dos atributos extraídos e a presença de técnicas de ofuscação (Bishop, 2006). No Capítulo 4, descreve-se o procedimento de validação cruzada, as métricas de avaliação (acurácia, precisão, revocação, F_1-score) e o protocolo experimental empregado para comparar os quatro classificadores na tarefa de detecção de binários *wasm* maliciosos dedicados à mineração não autorizada de cripto moedas.

Algoritmo	**Interpretabilidade**	**Custo Computacional**	**Principal Limitação**
Regressão Logística	Alta	Muito baixo	Não captura não linearidades
Random Forest	Baixa	Médio	Baixa interpretabilidade
XGBoost	Muito baixa	Médio/Alto	Requer calibração fina
SVM Linear	Média	Baixo	Sensível a escala dos dados

Tabela 2.1: Comparação entre os algoritmos de classificação quanto à interpretabilidade, custo computacional e principais limitações.

Embora os algoritmos de *Machine Learning* apresentados especialmente *Random Forest* e *XGBoost* demonstrem bom desempenho em problemas com dados estruturados, eles dependem fundamentalmente de engenharia manual de características. Em contraste, os métodos de *Deep Learning*, discutidos na próxima seção, são capazes de aprender representações hierárquicas automaticamente a partir de dados brutos, sendo particularmente eficazes em tarefas envolvendo imagens, áudio, texto e sinais temporais (LeCun et al., 2015).

2.4.3 Deep Learning

A Aprendizagem Profunda - DL (*Deep Learning*) é uma subárea central do *Machine Learning*. Formalmente, é definido como um conjunto de algoritmos que empregam redes neuronais artificiais com múltiplas camadas de processamento não linear para a extração, transformação e modelação de representações de dados com elevados níveis de abstração (Pouyanfar et al., 2018). Diferentemente dos algoritmos tradicionais que frequentemente requerem extração manual de características (*feature engineering*), os modelos de *Deep Learning* são capazes de aprender representações hierárquicas diretamente dos dados brutos.

Matematicamente, uma rede neuronal profunda é uma composição de funções não lineares. O processamento da informação numa camada oculta l pode ser definido pela seguinte equação, essencial para compreender a propagação dos dados (Pouyanfar et al., 2018):

$$\mathbf{h}^{(l)} = \sigma\left(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}\right), \tag{2.3}$$

onde:

- $\mathbf{h}^{(l)}$ é o vetor de ativação (saída) da camada atual l;
- $\mathbf{h}^{(l-1)}$ é o vetor de entrada proveniente da camada anterior (ou os dados brutos, se $l = 1$);
- $\mathbf{W}^{(l)}$ é a matriz de pesos sinápticos da camada;
- $\mathbf{b}^{(l)}$ é o vetor de viés (*bias*);

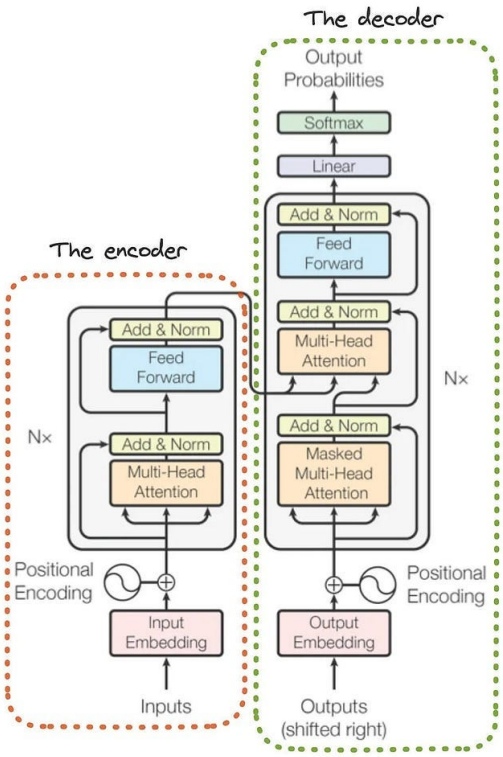
- σ é uma função de ativação não linear (como ReLU, Sigmoide ou Tanh) que permite à rede mapear relações complexas;

na cibersegurança, ao analisar representações de WebAssembly, as camadas mais profundas destas redes podem capturar padrões estruturais não lineares subtis que indicam intenções maliciosas, contornando técnicas comuns de ofuscação.

2.4.4 Arquitetura Transformers

Embora originalmente concebida para o Processamento de Linguagem Natural (NLP) (*Natural Language Processing*), a arquitetura de *Transformer* (Vaswani et al., 2017) consolidou-se como o estado da arte para a modelação de dados sequenciais complexos, como o *bytecode WebAssembly*. O código compilado em *WebAssembly* pode ser interpretado matematicamente como uma longa sequência de *tokens* discretos. Da mesma forma que os modelos de linguagem compreendem a semântica de uma frase, os modelos baseados em *Transformers* aprendem a "linguagem"do código executável (Bordawekar and Shmueli, 2017).

Figura 2.4: Arquitetura do Transformer, Fonte: Vaswani et al. (2017).



O núcleo inovador desta arquitetura é o mecanismo de auto atenção (*self-attention*), que permite ao modelo avaliar a sequência inteira em simultâneo, calculando um “peso” de importância para cada *token* em relação a todos os outros elementos da sequência (Vaswani et al., 2017). Diferentemente de RNNs ou CNNs, a auto atenção captura dependências de longo alcance sem sofrer com o desaparecimento de gradientes (Hochreiter, 1998; Bai et al., 2018). Ao ponderar globalmente as interações entre tokens, o modelo identifica padrões contextuais em scripts maliciosos, como ofuscações distribuídas ou chamadas perigosas dependentes de variáveis distantes. Essa capacidade confere à arquitetura maior expressividade e generalização para detecção de ameaças (Devlin et al., 2019; Dosovitskiy et al., 2021).

No contexto da cibersegurança e da análise de binários *wasm*, esta característica assume uma importância tática inestimável. Diferente de modelos clássicos que analisam apenas instruções vizinhas

(n-gramas), o *Transformer* consegue correlacionar uma importação de telemetria no início do módulo com um laço de repetição (*Loop*) ou uma operação de 64 bits (`i64.xor`) situados em blocos distantes. Esta capacidade de reconstruir o contexto semântico global do ataque permite gerar *embeddings* vetoriais altamente precisos. Como demonstrado por trabalhos recentes na área de vetorização (Peralta et al., 2025), a representação gerada por estes mecanismos de atenção facilita a fusão de diferentes fragmentos de dados num único espaço vetorial, tornando o modelo analítico resiliente a técnicas de evasão e ofuscação que fraturam a lógica do programa (Cabrera-Arteaga et al., 2023; Peralta et al., 2025).

2.4.5 Chunking

A análise de binários *wasm* resulta, frequentemente, na extração de artefatos de execução extremamente volumosos. Esta volumetria apresenta um desafio técnico; Dado que os modelos baseados em *Transformers* possuem restrições arquitetônicas rigorosas quanto à dimensão máxima da sequência de entrada. Para contornar esta limitação computacional, introduz-se a técnica de segmentação. Essa técnica consiste na divisão sistemática dos dados originais em partições menores *chunks*. A eficácia desta técnica exige um particionamento semântico, onde o binário *wasm* é dividido respeitando as suas fronteiras lógicas naturais (como funções ou blocos básicos), garantindo que o contexto de cada segmento é preservado (Taipalus et al., 2024).

Formalmente, considere-se um módulo de código representado como uma sequência de *tokens* $S = (x_1, x_2, \ldots, x_N)$. A operação de *chunking* mapeia a sequência S num conjunto de subsequências (ou *chunks*) $C = \{c_1, c_2, \ldots, c_M\}$. No método de janela deslizante (*sliding window*), com uma janela de tamanho L e um passo (*stride*) de tamanho P, o i-ésimo *chunk* é definido como:

$$\mathbf{c_i} = (\mathbf{x_{(i-1)P+1}}, \mathbf{x_{(i-1)P+2}}, \ldots, \mathbf{x_{(i-1)P+L}}), \tag{2.4}$$

onde $1 \leq i \leq M$ e $M = \lfloor \frac{N-L}{P} \rfloor + 1$. Para que não haja perda de contexto semântico, é comum configurar $P < L$, gerando uma sobreposição entre os *chunks* (Feng et al., 2020).

Após o processamento individual de cada *chunk*, é possível aplicar métodos de fusão para gerar uma representação consolidada do ficheiro (Peralta et al., 2025) ou, alternativamente, indexar os vetores de cada segmento de forma independente no VDBMS. Esta granularidade viabiliza a execução de pesquisas de similaridade ao nível do bloco de código (Pan et al., 2023), o que permite a identificação de segmentos associados a sub-rotinas maliciosas inseridas em módulos predominantemente benignos (Taipalus, 2024).

2.4.6 Representação Semântica

A representação semântica constitui o alicerce sobre o qual operam os sistemas de busca por similaridade em espaços vetoriais. Trata-se do processo de extração de elementos discretos como palavras, documentos ou, no contexto desta pesquisa, segmentos de código para vetores contínuos em um espaço de alta dimensionalidade (Mikolov et al., 2013a). Esses vetores, denominados *embeddings*, são projetados de modo que a proximidade geométrica no espaço latente reflita a similaridade semântica entre os elementos originais.

Embeddings de Código

A aplicação de embeddings à análise de código baseia-se no princípio de que o código executável, assim como a linguagem natural, possui uma estrutura composicional e padrões recorrentes que podem ser capturados por modelos de Aprendizado de Máquina (Bordawekar and Shmueli, 2017). Diferentemente da representação textual convencional, que trata cada *token* como um símbolo discreto, os *embeddings*

projetam *tokens* ou sequências de *tokens* em um espaço contínuo onde relações de similaridade são representadas por distâncias geométricas (Pan et al., 2023).

Formalmente, um *embedding* é uma função $f : \mathcal{X} \to \mathbb{R}^D$ que mapeia um elemento $x \in \mathcal{X}$ um *token*, uma instrução, um bloco de código para um vetor $\mathbf{v} \in \mathbb{R}^D$ de dimensão D. O espaço resultante $\mathbb{R}^D$, frequentemente denominado *espaço latente*, organiza os elementos de modo que a distância entre vetores correlacione-se com a similaridade semântica entre os elementos correspondentes (Taipalus, 2024).

Propriedades do Espaço Latente

No espaço latente, a similaridade comportamental entre amostras de código manifesta-se por meio da proximidade geométrica. Módulos *WebAssembly* que compartilham lógicas de execução análogas mesmo quando submetidos a técnicas de ofuscação sintática tendem a ser projetados em regiões próximas do espaço vetorial. Em contrapartida, códigos benignos e maliciosos, por apresentarem comportamentos fundamentalmente distintos, ocupam regiões separadas (Taipalus, 2024).

Esta propriedade confere à representação semântica uma vantagem crucial para a detecção de *malware*: a capacidade de generalização. Diferentemente de abordagens baseadas em assinaturas, que falham diante de variantes previamente desconhecidas, um classificador que opera sobre *embeddings* pode identificar padrões maliciosos emergentes com base na proximidade vetorial a exemplos conhecidos (Ahmadi, 2025).

Operacionalização da Similaridade

Para que as representações semânticas sejam aplicáveis em sistemas de detecção, é fundamental definir métricas que quantifiquem a proximidade entre os vetores. A Similaridade de Cosseno, detalhada na Seção 2.5.1, é amplamente utilizada para este fim. A sua invariância à magnitude vetorial permite que a análise se concentre exclusivamente na orientação espacial, uma característica essencial quando se comparam segmentos de código de diferentes extensões (Pan et al., 2023). Consequentemente, a transposição da detecção de *malware* para o domínio vetorial converte a identificação de comportamentos maliciosos num problema de separação topológica. Sob esta formulação, espera-se que amostras benignas e maliciosas ocupem regiões distintas no espaço latente, assegurando a eficácia de classificadores baseados em proximidade, como o algoritmo k-NN, discutido na Seção 2.5.2 (Taipalus, 2024; Berrios et al., 2025).

2.4.7 Modelos de Linguagem de Grande Escala na Análise de Código

Os LLMs têm se consolidado como uma abordagem promissora para a análise semântica de código, estendendo o alcance das técnicas tradicionais ao representar relações contextuais entre instruções e estruturas de programas (She et al., 2024; Huang and Zhao, 2024). Na segurança da informação, esses modelos têm sido aplicados tanto à classificação automática de *malware* quanto à geração de explicações interpretáveis (Marais et al., 2025; Jelodar et al., 2025). A aplicação de LLMs à análise de código apresenta vantagens significativas. Primeiramente, a capacidade de interpretar estruturas formais como ASTs e CFGs permite a compreensão de padrões estruturais em código fonte e binários (Ma et al., 2023). Adicionalmente, esses modelos demonstram capacidade de generalização a partir de poucos exemplos (Mann et al., 2020), desenvolvem capacidades, habilidades não programadas explicitamente que surgem naturalmente com o aumento da escala do modelo, como raciocínio contextual avançado (Wei et al., 2022) e permitem a reutilização do mesmo modelo em múltiplas tarefas (Bommasani, 2021).

Apesar do potencial, persistem desafios significativos. Estudos apontam que os modelos compreendem de forma limitada a semântica dinâmica e o comportamento em tempo de execução, o que pode levar a interpretações equivocadas, fenômeno conhecido como alucinação semântica (Ma et al., 2023). Adicionalmente, os modelos são vulneráveis a ataques adversariais e podem ser utilizados para geração

de variantes maliciosas (Jelodar et al., 2025). Outro fator crítico é a concentração do desenvolvimento em poucas famílias de modelos, criando dependências tecnológicas relevantes (Bommasani, 2021). A literatura recomenda, portanto, abordagens híbridas que combinem a generalização dos LLMs com verificações formais e técnicas tradicionais de análise programática (Ma et al., 2023).

No contexto desta pesquisa, as características dos LLMs são particularmente relevantes para a detecção de *cryptojacking* em *WebAssembly*. A capacidade de interpretar ASTs permite reconhecer padrões típicos de mineração, como chamadas indiretas a funções de *hash*, operações intensivas de memória e *loops* repetitivos (Loose et al., 2023). Quando associados a uma catalogação sistemática de dados incluindo metadados como tipos de instrução, fluxos de memória e taxonomias como MITRE ATT&CK os LLMs oferecem um quadro promissor para correlacionar sinais sintáticos e semânticos, ampliando a capacidade de identificar ataques que exploram *WebAssembly* e JavaScript para mineração ilícita de cripto moedas (Marais et al., 2025; Jelodar et al., 2025).

2.5 Banco de Dados Vetoriais

Um Sistema de Gerenciamento de Bancos de Dados Vetoriais (VDBMS) é uma infraestrutura especializada, projetada para armazenar, gerenciar e recuperar dados de alta dimensionalidade representados como vetores em um espaço contínuo (Taipalus, 2024). Diferentemente dos Sistemas de Gerenciamento de Bancos de Dados Relacionais (RDBMS), que recuperam informações por meio de correspondências exatas e lógica booleana sobre dados tabulares, o VDBMS fundamenta-se no paradigma da busca por similaridade. Nesse modelo, o sistema processa consultas ($\mathbf{q}$) utilizando métricas de distância geométrica como a similaridade de cosseno ou a distância euclidiana para identificar os vetores mais próximos no espaço de representação, operação frequentemente viabilizada por algoritmos de Busca de Vizinhos Mais Próximos (k-NN) ou suas aproximações escaláveis (ANNS) (Pan et al., 2023).

A arquitetura de um VDBMS moderno é otimizada para contornar a degradação de desempenho inerente à alta dimensionalidade. Isso é alcançado por meio de estruturas de indexação avançadas, como grafos navegáveis hierárquicos HNSW *(Hierarchical Navigable Small World)*, que garantem a viabilidade computacional em buscas de larga escala. Além do desempenho na recuperação, um VDBMS robusto fornece garantias transacionais e operacionais, como escalabilidade, controle de concorrência e a capacidade crucial de associar metadados determinísticos (denominados payloads) a cada vetor indexado (Taipalus, 2024).

É precisamente essa capacidade de integrar representações não estruturadas a metadados estruturados que torna o VDBMS essencial no domínio da segurança computacional. Sistemas de detecção de malware baseados exclusivamente em embeddings semânticos (como representações neurais de código) são comprovadamente vulneráveis a técnicas de ofuscação, que alteram a superfície sintática do binário sem modificar seu comportamento malicioso subjacente (Loose et al., 2023; Jelodar et al., 2025). Para superar essa fragilidade metodológica, o VDBMS viabiliza a operacionalização de espaços vetoriais. Formalmente, o sistema pode gerenciar um espaço enriquecido unificando um vetor de características semânticas $\mathbf{s} \in \mathbb{R}^{D_s}$ e um vetor de características estruturais dinâmicas ou estáticas $\mathbf{t} \in \mathbb{R}^{D_t}$. Essa unificação pode ocorrer por meio da concatenação direta, resultando no vetor $\mathbf{h}$:

$$\mathbf{h} = \mathbf{s} \oplus \mathbf{t} = [s_1, s_2, \ldots, s_{D_s}, t_1, t_2, \ldots, t_{D_t}] \in \mathbb{R}.^{D_s+D_t}. \tag{2.5}$$

Alternativamente, VDBMS avançados permitem a gestão desses dados por meio da separação de componentes, onde a semântica abstrata ($\mathbf{s}$) é tratada no espaço geométrico, enquanto as métricas estruturais e estatísticas de fluxo de controle ($\mathbf{t}$) atuam como filtros booleanos rígidos no payload (Ma et al., 2023). Independentemente da estratégia de fusão adotada, a utilização de um VDBMS permite

que o modelo resultante avalie simultaneamente relações contextuais profundas e regras determinísticas rigorosas. Consequentemente, o sistema de recuperação torna-se altamente preciso em cenários controlados e resiliente diante de *malwares* polimórficos ou adversarias (Ahmadi, 2025; Berrios et al., 2025).

2.5.1 Métricas de Similaridade

A recuperação de vetores em um VDBMS fundamenta-se em funções matemáticas que quantificam a proximidade geométrica entre vetores em um espaço de alta dimensionalidade. Formalmente, uma métrica de comparação é uma função $d : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ que mapeia dois vetores a um valor escalar. Dependendo da natureza da função, ela pode atuar como uma medida de similaridade (onde valores mais altos indicam maior proximidade) ou como uma métrica de distância (onde valores mais baixos, próximos a zero, indicam maior semelhança) (Pan et al., 2023).

As métricas mais amplamente adotadas em sistemas de busca vetorial incluem:

Similaridade de Cosseno: Mede o cosseno do ângulo entre dois vetores, focando na orientação e sendo invariante à magnitude deles. É dada por:

$$\text{cosine}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|} = \frac{\sum_{i=1}^{D} a_i b_i}{\sqrt{\sum_{i=1}^{D} a_i^2}\sqrt{\sum_{i=1}^{D} b_i^2}}, \tag{2.6}$$

essa métrica é particularmente adequada para representações textuais e de código, onde a direção do vetor no espaço latente reflete propriedades semânticas essenciais, enquanto a magnitude pode variar apenas em função do tamanho da entrada (Taipalus, 2024).

Distância Euclidiana (L_2): Calcula a distância geométrica direta em linha reta entre dois pontos no espaço vetorial, sendo altamente sensível à magnitude dos vetores. É definida pela equação:

$$\text{euclidean}(\mathbf{a}, \mathbf{b}) = \sqrt{\sum_{i=1}^{D} (a_i - b_i)^2}, \tag{2.7}$$

como se trata de uma métrica de distância, valores menores indicam maior semelhança. É a escolha apropriada quando o comprimento do vetor (magnitude) carrega informação estrutural ou comportamental relevante para a tarefa de classificação (Pan et al., 2023).

Produto Interno (Dot Product): Calcula a soma dos produtos elemento a elemento entre os dois vetores:

$$\text{dot}(\mathbf{a}, \mathbf{b}) = \sum_{i=1}^{D} a_i b_i, \tag{2.8}$$

trata-se de uma métrica computacionalmente muito eficiente. Quando os vetores são previamente normalizados para possuírem comprimento igual a 1, o Produto Interno torna-se matematicamente equivalente à Similaridade de Cosseno (Taipalus, 2024)

A escolha da métrica impacta diretamente a eficácia da recuperação e deve estar estritamente alinhada às características dos dados e ao domínio de aplicação (Taipalus, 2024). Para representações semânticas geradas por modelos de Aprendizado de Máquina, a Similaridade de Cosseno é amplamente recomendada, pois normaliza os efeitos de escala e foca na abstração latente dos vetores (Pan et al., 2023). No contexto desta pesquisa, onde representações vetoriais de código *WebAssembly* são comparadas para a identificação de padrões maliciosos, a Similaridade de Cosseno mostra-se particularmente adequada. Diferentes instâncias de um ataque de *cryptojacking* podem apresentar variações significativas no volume de instruções alterando a magnitude do vetor, mas tendem a preservar a direção geométrica associada à sua semântica comportamental maliciosa.

2.5.2 Algoritmos de Indexação

A viabilidade operacional da busca por similaridade em espaços de alta dimensionalidade é assegurada por algoritmos de indexação que organizam os vetores em estruturas de dados topologicamente otimizadas (Pan et al., 2023). O problema computacional subjacente é formalizado como a busca pelos k-Vizinhos Mais Próximos (k-NN). Em uma abordagem de força bruta, a busca exata requer a computação da distância entre o vetor de consulta e todos os vetores da base, resultando em uma complexidade temporal de $\mathcal{O}(N \cdot D)$, onde N é o número de amostras e D a dimensionalidade. Este custo linear impõe um gargalo computacional intransponível para bases de dados em larga escala (Taipalus, 2024).

Para mitigar essa limitação, os VDBMS modernos empregam métodos de Busca Aproximada por Vizinhos Mais Próximos (*Approximate Nearest Neighbor* - ANN). Esses algoritmos abdicam da garantia matemática de encontrar o vizinho exato em favor de uma recuperação probabilística altamente escalável, trocando frações de precisão (*recall*) por ordens de grandeza em velocidade. Embora existam diversas abordagens heurísticas como o particionamento do espaço com *Inverted File Index* (IVF) ou a compressão vetorial via *Product Quantization* (PQ) (Jegou et al., 2010), o estado da arte para buscas de baixa latência baseia-se em grafos.

Nesse contexto, destaca-se o algoritmo HNSW (*Hierarchical Navigable Small World*), que se consolidou como o padrão arquitetônico na maioria dos sistemas vetoriais avançados (Malkov and Yashunin, 2018). O HNSW organiza o espaço vetorial em um grafo hierárquico de múltiplas camadas, inspirado no conceito de redes de "mundo pequeno"(*small-world networks*) e *skip-lists*. Durante a indexação, conexões longas são estabelecidas nas camadas superiores para permitir grandes saltos pelo espaço vetorial, enquanto conexões curtas e densas nas camadas inferiores garantem a exploração refinada da vizinhança local. Durante a busca, o algoritmo emprega um roteamento guloso (*greedy routing*), descendo pelas camadas até encontrar o mínimo local, reduzindo a complexidade temporal para níveis logarítmicos $\mathcal{O}(\log N)$.

A infraestrutura gerada pelo HNSW é fundamental não apenas para a recuperação, mas para o suporte à inferência analítica. O conceito de k-NN transcende a busca geométrica e é frequentemente utilizado como um classificador não paramétrico, no qual a classe de um novo elemento é inferida pela votação majoritária entre seus k vizinhos mais próximos recuperados do índice (Géron, 2022).

Nesta pesquisa, a eficiência de indexação do HNSW é o mecanismo que viabiliza a classificação em tempo hábil dos *embeddings* de *WebAssembly*. Ao mapear o código binário para um espaço estruturado por esse grafo, torna-se possível identificar comportamentos maliciosos como instâncias de *cryptojacking* de forma quase instantânea, garantindo que o tempo de resposta do sistema seja compatível com os requisitos de detecção de ameaças em ambientes de produção.

2.6 Qdrant: Arquitetura e funcionamento

O Qdrant é um Sistema de Gestão de Base de Dados Vetorial - VDBMS de código aberto, concebido sob o paradigma de processamento de alto desempenho para armazenamento, indexação e recuperação topológica de representações latentes de alta dimensionalidade (Qdrant Tech, 2026). Implementado na linguagem Rust, o sistema explora abstrações de custo zero (*zero-cost abstractions*) e segurança de memória em tempo de compilação (*memory safety without garbage collection*), características que mitigam *overheads* de concorrência e garantem latência estrita, consolidando-o como infraestrutura preferencial para arquiteturas de inferência de Inteligência Artificial em tempo real (Pan et al., 2023).

2.6.1 Arquitetura de Indexação e Modelagem Espacial

O núcleo do mecanismo de persistência do Qdrant repousa no desacoplamento arquitetural entre o armazenamento da carga útil transacional e as estruturas de indexação geométrica. Para a recuperação

por similaridade, o sistema implementa o algoritmo HNSW, uma estrutura topológica baseada em grafos multicamada que roteia consultas heuristicamente em espaços não-euclidianos. Este método de Busca Aproximada por Vizinhos Mais Próximos (ANN) reduz a complexidade temporal da inferência vetorial para limites logarítmicos, sustentando o equilíbrio assintótico entre a precisão de recuperação (*recall*) e o custo computacional (Malkov and Yashunin, 2018).

O diferencial arquitetônico do Qdrant reside na sua instrumentalização primária do modelo de espaços vetoriais, operacionalizado não apenas via matemática linear, mas mediante duas primitivas estruturais avançadas:

- ***Payloads* e Filtros de Cardinalidade:** O Qdrant suporta a anexação de metadados estruturados (o vetor determinístico $\mathbf{t}$) gerenciados por índices invertidos concorrentes. O otimizador heurístico de consultas avalia a cardinalidade e a seletividade destes predicados para injetar restrições booleanas rígidas diretamente no algoritmo de travessia do HNSW (*single-stage pre-filtering*). Essa técnica evade a exploração computacional de subgrafos irrelevantes antes do cálculo vetorial estrito de métricas de similaridade.

- ***Named Vectors* (Multirrepresentação Latente):** A arquitetura transcende a necessidade de concatenação física inicial ($\mathbf{s} \oplus \mathbf{t}$) ao permitir o mapeamento de múltiplos tensores ortogonais a um único identificador escalar. Esta abstração viabiliza a avaliação simultânea da semântica comportamental e de projeções estruturais em subespaços latentes distintos, otimizando a dimensionalidade efetiva durante a execução da consulta (Qdrant Tech, 2026).

2.6.2 Posicionamento no Ecossistema de VDBMS

O ecossistema contemporâneo de bancos de dados vetoriais segrega-se em paradigmas arquitetônicos divergentes. O panorama inclui desde plataformas orientadas ao *sharding* massivo (*e.g.*, Milvus) e abstrações fortemente acopladas a modelos generativos (Weaviate e Chroma), até motores de busca de classe corporativa (Vespa) e extensões que injetam capacidades latentes em sistemas relacionais tradicionais (pgvector). Em contraste com essas abordagens, o Qdrant consolida-se como um motor focado na colocalização em memória de dados vetoriais e índices escalares visando maximizar o *throughput* e o controle de consultas híbridas complexas (Pan et al., 2023).

A Tabela 2.2 sintetiza as restrições e topologias dos sistemas mais representativos na literatura e na indústria:

Sistema	**Core Engine**	**Topologia de Índice**	**Paradigma Arquitetônico Dominante**
Qdrant	Rust	HNSW	Predicados de estágio único (*Payload*)
Milvus	C++/Go	IVF/HNSW	Escalabilidade distribuída (*Cloud-native*)
Weaviate	Go	HNSW	Abstração semântica LLM integrada
Pinecone	Proprietário	HNSW	Infraestrutura unificada gerenciada (*DBaaS*)
Chroma	Rust/Python	HNSW	Prototipação local ágil (*AI-native*)
pgvector	C (PostgreSQL)	HNSW/IVFFlat	Extensão relacional com garantias ACID
Vespa	C++/Java	HNSW/Tensor	Processamento e motores de recomendação

Tabela 2.2: Comparação arquitetural entre os principais Sistemas de Gerenciamento de Bancos de Dados Vetoriais.

2.6.3 Aplicações em Segurança da Informação

Na área de segurança da informação, o Qdrant tem sido empregado em aplicações que exigem busca por similaridade em larga escala, como detecção de *malware* baseada em *embeddings*, sistemas de detecção de intrusão e análise de tráfego de rede (Taipalus, 2024). Sua capacidade de combinar busca vetorial com filtros de metadados permite, por exemplo, restringir a busca a amostras de uma determinada família de *malware* ou a um intervalo temporal específico, aumentando a precisão da detecção (Pan et al., 2023).

A escolha de um VDBMS para aplicações de segurança deve considerar requisitos como latência, volume de dados, necessidade de consultas híbridas e requisitos operacionais (Taipalus, 2024). O Qdrant apresenta-se como uma opção adequada para cenários que demandam baixa latência, suporte a filtros complexos e simplicidade de implantação.

2.7 Considerações Finais do Capítulo

Este capítulo apresentou os fundamentos teóricos que sustentam a proposta desta dissertação, organizados em torno de três eixos principais: a tecnologia *WebAssembly* e suas características arquiteturais, o fenômeno do *cryptojacking* como ameaça emergente no ecossistema web, e os métodos de análise e detecção baseados em aprendizado de máquina e bancos de dados vetoriais.

Inicialmente, foram descritas as características arquiteturais do *WebAssembly*, incluindo seu modelo de segurança baseado em sandbox, sua estrutura binária modular e sua capacidade de interoperabilidade com JavaScript. Embora essas propriedades viabilizem aplicações de alto desempenho no navegador, também criam oportunidades para exploração maliciosa, especialmente em ataques de *cryptojacking*, que utilizam o poder computacional do usuário sem seu consentimento para mineração de criptomoedas.

Em seguida, foram analisadas as principais abordagens de análise de binários *WebAssembly*, destacando-se a análise estática, dinâmica e híbrida. A análise estática, embora eficiente, mostra-se vulnerável a técnicas de ofuscação e diversificação de código. A análise dinâmica, por sua vez, oferece maior robustez diante dessas técnicas, mas impõe custos computacionais significativos. A análise híbrida emerge como uma alternativa promissora, combinando a eficiência da análise estática com a profundidade da análise dinâmica.

No âmbito do aprendizado de máquina, foram apresentados os fundamentos de LLMs e sua aplicação à análise de código. A arquitetura Transformer, com seu mecanismo de auto atenção, demonstra capacidade de capturar relações contextuais em sequências de código, gerando representações semânticas *embeddings* que refletem similaridades comportamentais entre amostras. No entanto, a dependência exclusiva de *embeddings* semânticos apresenta vulnerabilidades a ataques adversarias, motivando a exploração de abordagens híbridas que combinam múltiplas fontes de características.

Complementarmente, foram discutidos os conceitos de espaços vetoriais, que concatenam representações semânticas com características determinísticas extraídas da AST como contagens de operações aritméticas de 64 bits, acessos à memória e estruturas de repetição. Essa composição visa aumentar a robustez do classificador diante de técnicas de ofuscação que poderiam enganar abordagens baseadas exclusivamente em uma única fonte.

Por fim, foram apresentados os fundamentos dos Sistemas de Gestão de Base de Dados Vetoriais VDBMS, com ênfase no Qdrant, e discutidas as métricas de similaridade, os algoritmos de indexação aproximada HNSW, IVF, PQ e as técnicas de redução de dimensionalidade PCA, t-SNE. Esses elementos são essenciais para operacionalizar a busca por similaridade em larga escala e para a validação visual da separabilidade entre classes no espaço vetorial.

Em síntese, os fundamentos consolidados neste capítulo fornecem a base conceitual necessária para a definição da metodologia de pesquisa e para a elaboração da arquitetura proposta. O próximo capítulo

apresenta e analisa os trabalhos relacionados, situando a contribuição desta dissertação no contexto da literatura existente e identificando as lacunas que motivam a investigação proposta.

Capítulo 3

Trabalhos Relacionados

Este capítulo estabelece o arcabouço referencial da pesquisa por meio da análise do estado da arte concernente à detecção e mitigação de *cryptojacking*, com foco central na inspeção analítica de módulos binários compilados para *WebAssembly* (*wasm*). A taxonomia do parasitismo computacional em ambientes *web* originou-se paralelamente à consolidação dos protocolos de consenso baseados em Prova de Trabalho (*Proof-of-Work*), inicialmente operacionalizada via *scripts* interpretados, projetados para o sequestro furtivo de ciclos de processamento do hospedeiro (Eskandari et al., 2018; Hong et al., 2018).

A padronização do *WebAssembly* pelo W3C subverteu esse modelo de ameaça clássico, catalisando uma transição morfológica drástica nos artefatos de *malware*. Ao fornecer um formato de instrução binária de baixo nível, executável em uma *sandbox* baseada em pilha com desempenho assintótico ao código nativo (via compilação *Just-In-Time* ou *Ahead-of-Time*), o *wasm* emergiu como um vetor de exploração altamente resiliente. Essa arquitetura estrita permite que rotinas criptográficas intensivas contornem gargalos de interpretação tradicionais e evadam mecanismos de detecção heurística baseados em assinaturas textuais (Lonkar and Chandrayan, 2018; Cabrera-Arteaga et al., 2023). Consequentemente, a literatura científica ramificou-se em três eixos metodológicos de instrumentação: estática, dinâmica e híbrida (Almurshid et al., 2024).

A análise estática fundamenta-se na extração de características morfológicas como topologia de CFG e distribuições de entropia de *opcodes* diretamente do binário bruto, dispensando a instanciação do módulo em memória. Apesar de sua incontestável eficiência computacional, esta vertente apresenta degradação severa de *recall* quando confrontada com metamorfismo sintático, inserção de código morto e ofuscação algorítmica, técnicas que fragmentam a superfície estrutural sem alterar a semântica maliciosa latente. Em antítese, a análise dinâmica instrumentaliza o ambiente de execução para extrair a telemetria comportamental (*e.g.*, *profiling* de *heap*, interrupções de CPU e chamadas atípicas à API de *Web Workers*). Embora semanticamente robusta à ofuscação, esta abordagem introduz *overheads* inaceitáveis de latência e permanece suscetível a técnicas de evasão ambiental (*sandbox evasion*) e inatividade condicional (*logic bombs*).

Para reconciliar a escalabilidade da inspeção estática com a fidelidade semântica da instrumentação dinâmica, abordagens híbridas consolidaram-se como a fronteira de pesquisa contemporânea. Sob essa perspectiva analítica, as seções subsequentes deste capítulo escrutinam as arquiteturas de detecção publicadas recentemente. O objetivo excede o mapeamento cronológico, focando na identificação sistemática das limitações estruturais destas propostas notadamente no que tange ao processamento escalável de abstrações de alta dimensionalidade, evidenciando a lacuna metodológica (*research gap*) que justifica a proposição da infraestrutura baseada em VDBMS explorada nesta dissertação.

3.1 Revisão Bibliográfica

Esta seção estabelece o arcabouço referencial da pesquisa por meio da análise do estado da arte concernente à detecção de *cryptojacking* também tipificado como *drive-by mining* operacionalizado via módulos *WebAssembly* incorporados em ambientes *web*. O foco diretivo recai sobre a avaliação da eficácia e da escalabilidade das arquiteturas de detecção propostas na literatura recente para mitigar essa classe de ameaça.

Além de mapear cronologicamente e caracterizar as abordagens de defesa existentes, esta revisão sistemática tem como propósito primordial a identificação de lacunas metodológicas e infraestruturais (*research gaps*). Ao contrastar as propostas atuais com os desafios impostos pela ofuscação algorítmica e pelo processamento de dados de alta dimensionalidade, esta seção fundamenta as direções para o desenvolvimento da arquitetura de detecção híbrida e vetorial proposta nesta dissertação.

3.1.1 Protocolo

A condução deste mapeamento bibliográfico foi norteada pelas diretrizes metodológicas de revisão sistemática estabelecidas por Kitchenham et al. (2007), integradas à estrutura de definição de objetivos do paradigma GQM (*Goal, Question, Metric*) concebido por Caldiera and Rombach (1994). A estratégia de busca delimitou-se ao recorte temporal de 2015 a 2025, indexando produções científicas focadas na interseção entre parasitismo computacional em navegadores, compilação *WebAssembly* e instrumentação de segurança em *malware*.

A formalização do escopo da revisão, bem como o alinhamento dos critérios de análise metodológica, encontram-se sintetizados na Tabela 3.1.

ANALISAR	Publicações científicas e literatura cinzenta primária.
COM O PROPÓSITO DE	Sistematizar as técnicas, caracterizar vulnerabilidades e analisar métricas de eficácia.
EM RELAÇÃO A	Estudos arquiteturais e algorítmicos voltados à detecção e classificação de campanhas de *cryptojacking* em *wasm*.
DO PONTO DE VISTA DOS	Pesquisadores da área de Segurança da Informação e Análise de *Malware*.
NO CONTEXTO	Acadêmico e de Engenharia de Software.

Tabela 3.1: Definição do Objetivo de Revisão Conforme o Paradigma GQM.

A partir da estruturação dos objetivos norteadores, a extração de dados da literatura foi direcionada por três Questões de Pesquisa (QP) fundamentais, formuladas para sequenciar a progressão analítica do problema à solução:

- **QP-1.** Quais são as principais abordagens morfológicas e técnicas de ofuscação (metamorfismo sintático) empregadas por agentes maliciosos para evadir mecanismos de detecção em campanhas de *cryptojacking* baseadas em *WebAssembly*?

- **QP-2.** Quais arquiteturas de detecção (estáticas, dinâmicas ou híbridas) compõem o estado da arte para a mitigação desta ameaça, e quais são as limitações metodológicas inerentes a essas propostas, especialmente em relação a escalabilidade e *overhead* computacional?

- **QP-3.** Como a extração de características estruturais (*e.g.*, AST) e a geração de representações latentes densas (*embeddings*) têm sido instrumentalizadas para a identificação de similaridade de código e classificação escalável de *malware*?

O encadeamento destas questões suporta diretamente o desenvolvimento deste trabalho. A **QP-1** destina-se a mapear a evolução das técnicas adversariais, justificando a insuficiência das análises heurísticas tradicionais. A **QP-2** visa sistematizar o estado da arte em defesas, evidenciando que, embora as abordagens atuais detectem a ameaça, elas frequentemente falham em escalabilidade sistêmica. Por fim, a **QP-3** isola as tecnologias subjacentes ao modelo proposto nesta pesquisa (AST, semântica vetorial e busca baseada em VDBMS), pavimentando a justificativa técnica para a inovação arquitetural que será detalhada nos capítulos subsequentes.

3.1.2 Estratégia de Busca

A estratégia de busca adotada nesta revisão sistemática contempla dois elementos centrais: (*i*) a seleção das bases de dados acadêmicas; e (*ii*) a definição do tipo de publicação a ser incluído. Com relação às fontes de informação, foram selecionadas quatro bases amplamente reconhecidas na área de Ciência da Computação: IEEE Xplore[1], ACM Digital Library[2], Springer Link[3] e Scopus[4]. Essas plataformas foram escolhidas por sua cobertura abrangente de periódicos e conferências de alto impacto. Complementarmente, realizou-se uma busca manual com o objetivo de identificar trabalhos potencialmente relevantes não indexados nas bases consultadas.

Considerando o escopo internacional das publicações e a predominância do inglês como língua franca na área, todos os termos de busca e os critérios de seleção foram estabelecidos nesse idioma. No que se refere ao tipo de publicação, restringiu-se a seleção a artigos científicos completos publicados em periódicos e anais de conferências com revisão por pares. Essa delimitação visa assegurar o rigor metodológico e a qualidade dos estudos incluídos, uma vez que tais trabalhos são submetidos à avaliação de especialistas da área.

Para a construção da *string* de busca, foram utilizados os parâmetros do modelo PICOC (*Population, Intervention, Comparison, Outcome* e *Context*), conforme proposto por Petticrew and Roberts (2008). O elemento *Comparison* não foi aplicado, uma vez que esta revisão não tem como objetivo central a comparação quantitativa entre métodos concorrentes. Os demais parâmetros foram definidos da seguinte forma:

- ***Population* (Ambiente Alvo):** O ecossistema de execução das aplicações ("webassembly"OR "wasm"OR "wasm module"OR "wasm binary").
- ***Intervention* (Fenômeno/Ameaça):** O ataque alvo da investigação ("cryptojacking"OR "drive-by mining"OR "malicious mining"OR "browser-based mining"OR "malware"OR "in-browser mining"OR "web mining"OR "cryptomining malware").
- ***Context* (Tecnologias e Métodos):** As técnicas de extração de características e representação semântica utilizadas na arquitetura proposta ("embedding"OR "semantic representation"OR "vector database"OR "similarity search"OR "feature fusion"OR "large language model"OR "machine learning").
- ***Outcome* (Resultados Esperados):** O objetivo final da aplicação dos métodos ("analysis"OR "detection"OR "evaluation"OR "static analysis"OR "dynamic analysis"OR "code analysis").

A partir desses construtos, a estratégia de busca foi composta por uma *string* principal, formatada para aplicação nas bases Scopus, Springer Link e IEEE Xplore, estruturada pela conjunção dos parâmetros (*Population* AND *Intervention* AND (*Context* OR *Outcome*)):

[1] `https://ieeexplore.ieee.org/`
[2] `https://dl.acm.org/`
[3] `https://link.springer.com/`
[4] `https://www.scopus.com/`

(("webassembly"OR "wasm"OR "wasm module"OR "wasm binary") AND ("cryptojacking"OR "drive-by mining"OR "malicious mining"OR "browser-based mining"OR "mining malware"OR "in-browser mining"OR "web mining"OR "cryptomining malware") AND ("embedding"OR "semantic representation"OR "vector database"OR "similarity search"OR "feature fusion"OR "large language model"OR "machine learning"OR "analysis"OR "detection"OR "evaluation"OR "static analysis"OR "dynamic analysis"OR "code analysis"))

Com o propósito de assegurar a consistência metodológica e a relevância científica dos estudos incluídos nesta revisão sistemática, foram estabelecidos critérios explícitos de inclusão e exclusão, conforme descrito a seguir.

Código	**Critério de Inclusão (CI)**	**Código**	**Critério de Exclusão (CE)**
CI1	O artigo investiga ativamente ataques de *cryptojacking*, com ênfase em técnicas de ofuscação, evasão ou impacto no desempenho.	CE1	O artigo tangencia o tema, mas não tem como foco central a segurança em *WebAssembly* ou a detecção de malwares/criptomineradores.
CI2	O artigo propõe ou avalia abordagens (estáticas, dinâmicas ou híbridas) para a detecção de ameaças e malwares especificamente no ecossistema *WebAssembly*.	CE2	O texto completo não está disponível para consulta ou acesso integral nas bases de dados institucionais.
CI3	O artigo explora a extração de características estruturais (como AST), representações semânticas (*embeddings*) ou aprendizado de máquina aplicados à análise de código binário.	CE3	O artigo constitui duplicação, resumo expandido (*short paper*) ou versão preliminar de outro trabalho já incluído.
		CE4	O artigo encontra-se redigido em idioma distinto do inglês ou português.

Tabela 3.2: Critérios de Inclusão (CI) e Exclusão (CE) considerados na Revisão Sistemática.

Resultados da String de Busca

A busca realizada nas bases de dados resultou em um total inicial de 272 artigos (excluindo os retornos da ACM). Após a consolidação dos dados, os artigos foram analisados de forma criteriosa utilizando-se duas fases de filtragem. O 1º Filtro consistiu na remoção de registros duplicados resultantes da sobreposição de indexação entre as bases. O 2º Filtro consistiu na triagem por leitura de título e resumo (*abstract*), etapa na qual foram aplicados os quatro Critérios de Exclusão (CEs) para eliminação de falsos positivos, exigindo-se que os trabalhos remanescentes atendessem a pelo menos um dos três Critérios de Inclusão (CIs) definidos na metodologia. A distribuição quantitativa desse processo é apresentada na Tabela 3.3.

Bibliotecas Digitais	**Resultados da String**	**1º Filtro**	**2º Filtro**
IEEE Xplore	43	12	4
ACM Digital Library	573	45	11
Scopus	80	25	9
Springer Link	209	30	6
TOTAL	**905**	**112**	**30**

Tabela 3.3: Resultados retornados da string de busca e progressão das fases de triagem.

Na etapa inicial do processo de seleção, após a consolidação dos dados e a remoção dos registros duplicados (1º Filtro), procedeu-se ao exame rigoroso dos títulos, palavras-chave e resumos dos 112 artigos únicos recuperados nas bases consultadas (excluindo, neste escopo, os retornos da ACM). Para que um estudo fosse considerado elegível para a fase subsequente (2º Filtro), era estritamente necessário atender a pelo menos um dos três Critérios de Inclusão (CIs) previamente definidos, sem incorrer em nenhuma das condições limitantes estabelecidas pelos Critérios de Exclusão (CEs).

Na segunda etapa, realizou-se a leitura integral dos 30 artigos que satisfizeram os requisitos da triagem preliminar. A extração de dados foi conduzida com base na Ficha de Extração estruturada, concebida para fornecer suporte analítico rigoroso e responder diretamente às Questões de Pesquisa (QPs) delineadas. Esse procedimento assegurou a aplicação uniforme dos parâmetros de avaliação metodológica e técnica, promovendo a padronização da análise e facilitando a categorização sistemática dos resultados de cada estudo primário.

A estratégia de revisão adotada contribuiu para a identificação criteriosa dos trabalhos mais relevantes, garantindo rigor metodológico e transparência em todas as fases do funil de seleção. A aplicação dessas diretrizes consolidadas, aliada às *strings* de busca refinadas, resultou em uma amostra representativa e atualizada do estado da arte voltado à análise e detecção de ataques de *cryptojacking* no ecossistema *WebAssembly*. Esse contingente de estudos fornece uma base sólida para a análise das abordagens estáticas e dinâmicas predominantes, evidenciando suas limitações e justificando o potencial de novas arquiteturas de classificação, conforme será discutido na seção seguinte.

A extração de dados e o escrutínio dos 30 estudos primários selecionados permitiram não apenas mapear o estado da arte, mas também conduzir uma análise das lacunas metodológicas e arquitetônicas existentes. A avaliação de métricas de desempenho (como acurácia, falsos positivos e *overhead* computacional), aliada à análise das táticas de evasão, providenciou o rigor necessário para sustentar a investigação. A seguir, apresenta-se a síntese narrativa e exaustiva dos resultados, estruturada em resposta às Questões de Pesquisa (QPs) delineadas na metodologia.

QP-1. Técnicas de implementação e ofuscação em ataques de *cryptojacking* direcionados a *WebAssembly*

O panorama das ameaças de mineração não autorizada no ecossistema *web* sofreu uma metamorfose profunda nos últimos anos, impulsionada pela busca incessante dos atacantes por maior rentabilidade e evasão. A literatura académica demonstra que o *cryptojacking* deixou de ser um mero oportunismo baseado em *scripts* textuais para se tornar uma engenharia de exploração de recursos de alto desempenho. Esta evolução encontra-se intrinsecamente ligada à adoção do *wasm*) pelos navegadores modernos. Concebido como uma linguagem *assembly* de formato binário e uma máquina virtual baseada em pilha (*stack-based*), o Wasm oferece uma execução com velocidade quase nativa (*near-native speed*). Estas características providenciaram o ambiente de execução ideal para algoritmos criptográficos *memory-bound*, como o CryptoNight, que requerem operações *bitwise* intensivas e acesso rápido à memória para serem rentáveis (Musch et al., 2019). Consequentemente, o vetor de ameaça transitou de um cenário onde a inspeção de código e a sua árvore de execução DOM eram triviais e transparentes (no caso do JavaScript) para um paradigma de binários pré-compilados e altamente opacos. Neste novo contexto, os agentes maliciosos beneficiam de uma assimetria defensiva severa: enquanto as vítimas cedem silenciosamente poder computacional e recursos energéticos, os defensores deparam-se com uma barreira semântica intransponível, onde o *bytecode* malicioso se camufla perfeitamente entre módulos de aplicações *web* legítimas de processamento pesado, tais como motores de renderização 3D ou ferramentas de edição de vídeo.

Da Mineração em JavaScript aos binários WebAssembly

Historicamente, as campanhas de *cryptojacking* operavam de forma centralizada e dependiam de ficheiros JavaScript injetados em páginas comprometidas. O trabalho pioneiro de Eskandari et al. (2018) e a análise em larga escala conduzida por Hong et al. (2018) documentaram a ascensão de serviços *drop-in* como o Coinhive, que permitiam a qualquer administrador ou atacante rentabilizar o tráfego *web* através de algoritmos de *hashing* interpretados. Contudo, as limitações de desempenho do motor V8 do

JavaScript perante cálculos iterativos pesados forçaram a evolução. Com a padronização do Wasm, os atacantes passaram a compilar linguagens nativas C, C++, e Rust diretamente para módulos *.wasm*. Como documentado por Musch et al. (2019), esta transição permitiu paralelizar o cálculo de *hashes* através da API *WebWorkers*, multiplicando exponencialmente o *hashrate* (taxa de *hashes* por segundo) sem a penalização de tempo de compilação JIT que afetava os *scripts* legados.

Ofuscação Estrutural e Diversificação Binária

Com o encapsulamento das rotinas de mineração em módulos *wasm*, os atacantes aperfeiçoaram técnicas de ofuscação projetadas especificamente para este formato binário. Trabalhos como o de Cabrera-Arteaga et al. (2023) e a dissecação técnica apresentada por Bhansali et al. (2022) reportam que a diversificação binária automática é capaz de alterar a estrutura do *bytecode* compilado sem penalizar a capacidade de mineração. Os autores documentam técnicas agressivas, tais como o achatamento do fluxo de controlo (*control-flow flattening*) que insere blocos lógicos opacos e declarações *switch* gigantescas para destruir a linearidade da execução, a injeção massiva de código morto (*dead code insertion*) na secção de funções (*Function Section*) e a ofuscação matemática de *opcodes*. Estas mutações garantem que cada infeção produza um binário com um *hash* criptográfico único, tornando as abordagens de detecção baseadas em assinaturas estáticas comerciais (como o VirusTotal) virtualmente obsoletas e com taxas de falsos negativos insustentáveis.

Evasão em tempo de execução, throttling e fingerprinting

Para além das mutações a nível estrutural, a literatura recente revela uma acentuada sofisticação na evasão em tempo de execução. Os módulos Wasm maliciosos já não operam a 100% da capacidade do processador. O estudo de Bian et al. (2020) documenta a implementação de algoritmos de *throttling* dinâmico (estrangulamento programado do uso de CPU), onde o *malware* monitoriza a taxa de *frames* do navegador (*frames per second* - FPS) e suspende a mineração caso detete um abrandamento na renderização, contornando assim as heurísticas de monitorização do sistema operativo. Paralelamente, o trabalho de Saad and Mohaisen (2024) expõe a dissimulação das comunicações de rede. Os atacantes abandonaram os pedidos HTTP tradicionais, passando a utilizar WebSockets cifrados para comunicar interativamente com as *pools* de mineração. Esta comunicação inclui o envio de *fingerprinting* de *hardware* do dispositivo da vítima, permitindo ao servidor de Comando e Controlo (C2) enviar cargas de trabalho modulares ajustadas especificamente ao número de núcleos (*cores*) e arquitetura do processador infetado. Esta estratégia assegura a persistência silenciosa da ameaça, evitando o colapso do sistema e prolongando o tempo de permanência (*dwell time*) na máquina hospedeira.

QP-2. Abordagens propostas para a detecção e as suas limitações estruturais

A identificação de módulos de *cryptojacking* ocultos em tráfego *web* exige a monitorização contínua e a correlação de anomalias computacionais a diferentes níveis de abstração. Esta exigência forçou a comunidade científica a repensar a arquitetura das soluções de segurança em navegadores e ferramentas *host-based*. A literatura sistematizada revela que a defesa contra a mineração não autorizada tem operado num ciclo reativo e assimétrico. Inicialmente, as metodologias de detecção limitavam-se ao bloqueio de domínios em listas de interdição (*blocklists*) e à análise de metadados em *scripts* interpretados, abordagens que capitularam rapidamente perante as capacidades de polimorfismo do novo ecossistema. Com o advento e adoção universal do Wasm, os investigadores foram obrigados a descer ao nível do *hardware* e da semântica de baixo nível, bifurcando as soluções propostas em duas vertentes arquitetônicas principais: o perfilamento dinâmico baseado em Contadores de Desempenho de Hardware (*Hardware Performance*

Counters - HPCs) e a reconstrução semântica através de Representações Intermédias (IR). Contudo, a análise dos resultados experimentais dos estudos primários aponta unanimemente para um compromisso incontornável (*trade-off*): a elevada precisão na detecção de ameaças *zero-day* acarreta uma sobrecarga computacional (*overhead*) que inviabiliza a escalabilidade e a adoção destas ferramentas em cenários de navegação *web* realista, onde múltiplas abas e processos concorrem por recursos limitados.

Abordagens inicias baseadas em JavaScript

Antes de avançar para a complexidade binária, a literatura assentou alicerces importantes na análise do JavaScript, criando um paradigma que, embora datado, inspirou as defesas modernas. O sistema CMTracker (Hong et al., 2018) introduziu heurísticas de rastreio comportamental focadas na identificação de *loops* intensivos no motor V8, analisando a periodicidade de execução (*stack-based profiler*). O trabalho seminal do SEISMIC (Wang et al., 2018) adotou uma abordagem preventiva, onde monitores introduzidos de forma *in-line* no código interceptavam as APIs criptográficas para solicitar consentimento explícito do utilizador. De forma mais sofisticada, o Outguard (Kharraz et al., 2019) extraiu métricas de instâncias de WebWorkers e tráfego de rede para alimentar classificadores SVM. Apesar destas ferramentas atingirem taxas de detecção na ordem dos 97% no auge do Coinhive, elas possuíam uma vulnerabilidade estrutural: dependiam da legibilidade do código e da interação com o DOM. Perante módulos *wasm*, que são carregados como matrizes de *bytes* opacos (*ArrayBuffer*) e executados numa *sandbox* de memória linear sem tocar no DOM, estas heurísticas legadas perdem completamente a visibilidade do processo.

Análise Dinâmica e Contadores de Desempenho de Hardware (HPC)

A falha das ferramentas textuais deu lugar à análise baseada no comportamento físico da máquina. Soluções de vanguarda como o MineSweeper (Konoth et al., 2018) abandonaram a inspeção do *software* para se focarem no *hardware*. Este sistema baseia a sua detecção na recolha de métricas provenientes de HPCs, monitorizando a frequência atípica de *cache misses* (falhas na memória *cache* de nível L3) e o excesso de predições de saltos condicionais (*branch predictions*), anomalias geradas pelo processamento do algoritmo CryptoNight. Da mesma forma, o sistema MINOS* (Naseem et al., 2021) avalia o ciclo de vida dos *threads* instanciados pelos navegadores. Como a inspeção ocorre ao nível da camada de processamento do Sistema Operativo, ela é absolutamente agnóstica a qualquer técnica de ofuscação ao nível do *bytecode Wasm*. No entanto, os próprios autores (Wu et al., 2022; Perrone and Romano, 2025) reconhecem os estrangulamentos arquitetónicos deste modelo. A recolha contínua de telemetria HPC em navegadores modernos, que por si só instanciam múltiplos *threads* de renderização de GPU e *scripts* em paralelo, introduz não só um *overhead* de CPU indesejável (frequentemente superior a 12%), mas também resulta num volume elevado de falsos positivos quando o utilizador executa aplicações *web* lícitas de alto desempenho.

Reconstrução Semântica Estática e Mapeamento de Grafos

Para evitar a sobrecarga em tempo de execução, a comunidade propôs o retorno à análise estática, não baseada em texto, mas em semântica abstrata. O projeto MinerRay (Romano et al., 2020) exemplifica o auge desta abordagem: a ferramenta decompila o módulo *wasm* e transpõe as suas instruções para uma IR. Em seguida, aplica algoritmos de fatiamento de programa (*program slicing*) para gerar um CFG simplificado, visando isolar unicamente as sequências semânticas associadas às rodadas lógicas das funções de *hashing*. Outras metodologias ainda mais disruptivas, como a apresentada na arquitetura NoCrypto (Wang et al., 2023), propõem a tradução de padrões de acesso de memória e *opcodes* para um espaço de cores visuais (imagens RGB), submetendo posteriormente estas imagens a CNNs para classificação. Embora as taxas de F-Score destas soluções semânticas superem frequentemente os 95%,

mesmo perante amostras altamente ofuscadas, a sua adoção prática colapsa no quesito da escalabilidade. O custo computacional e temporal para extrair caminhos de execução complexos e reconstruir CFGs a partir de binários *wasm* de vários *megabytes* impossibilita a sua utilização como mecanismo preventivo na latência ultrabaixa exigida para o carregamento de uma página *web*.

QP-3. Extração Sintática, Semântica Latente e Fusão de Características

A notória vulnerabilidade das assinaturas estáticas à ofuscação estrutural aliada aos custos proibitivos em termos de *overhead* exigidos pela análise dinâmica criaram uma lacuna científica substancial no domínio da cibersegurança *web*. A literatura mais recente procura colmatar esta falha explorando o ponto de intersecção entre a análise formal de programas (*Program Analysis*) e os modelos de aprendizagem profunda (*Deep Learning*). A premissa fundamental desta intersecção baseia-se no facto de que a conversão da estrutura bruta dos binários para modelos hierárquicos como árvores sintáticas permite preservar a intencionalidade de execução do código e captar o seu comportamento invariável, tornando irrelevantes as mutações de superfície e a ofuscação cosmética introduzidas pelos atacantes. Todavia, a adoção imatura de modelos de ML tradicionais tem exposto uma nova superfície de ataque. A literatura especializada adverte que a utilização de classificadores baseados em matrizes de características unidimensionais cria um ecossistema inerentemente suscetível ao ataque adversarial. Face a isto, os estudos primários apontam de forma concludente para a necessidade imperiosa de abandonar as métricas determinísticas isoladas e transitar para espaços vetoriais multidimensionais de semântica latente. A transição para *embeddings* contínuos constitui o estado da arte na resolução da tensão entre precisão, resistência à ofuscação e escalabilidade computacional.

Extração de caraterísticas a partir da AST

O *WebAssembly*, pela sua natureza formalmente validada e fortemente tipada, é idealmente propenso à engenharia reversa para representações hierárquicas. O estudo basilar de sistematização (Harnes and Morrison, 2024b) postula que a conversão direta de binários *wasm* em AST isola e preserva a invariância lógica do *malware*. A investigação detalhada conduzida por Xia et al. (2023) reforça empiricamente esta premissa: os autores demonstraram que a extração de características determinísticas a partir de nós específicos da AST tais como a frequência de instruções aritméticas de 64 *bits*, padrões de ramificações do fluxo de controlo e declarações de tipos de retorno é suficientemente rica para agrupar amostras maliciosas. Este processo prescinde da necessidade de executar o código (*zero execution overhead*) e apresenta elevada resiliência, dado que as operações fulcrais de *hashing* criptográfico mantêm a sua geometria sintática na árvore, não podendo ser removidas do nó sem destruir a eficácia do ataque de mineração.

Limitações dos classificadores de ML

Contudo, a extração minuciosa de características da AST resolve apenas a componente de modelação do problema; o mecanismo de classificação apresenta fraquezas latentes. A dependência da literatura anterior em classificadores tradicionais de ML tais como *Random Forest*, SVM ou *Gradient Boosting* aplicados sobre vetores estáticos unidimensionais revelou-se uma fragilidade explorável. O inovador estudo de Loose et al. (2023) expõe sistematicamente esta vulnerabilidade. Através do desenvolvimento da ferramenta Madvex, os investigadores provaram que atacantes podem usar instrumentação adversarial para modificar a distribuição de características num binário benigno e injetá-lo cirurgicamente num *payload* malicioso. Ao introduzir seções de código morto ou funções legítimas sem alterar o resultado da mineração, os atacantes conseguem desestabilizar os limiares de decisão (*decision boundaries*) definidos pelos hiperplanos dos algoritmos de ML tradicionais, enganando a máquina e resultando em taxas alarmantes de falsos negativos

perante ameaças conhecidas.

Espaços vetoriais e embeddings

Para superar a suscetibilidade a perturbações adversariais e garantir uma precisão operacional escalável, a vanguarda científica aponta para a implementação de arquiteturas de fusão de características (*feature fusion*) projetadas em alta dimensionalidade. A literatura sugere a transição da contagem isolada de características estáticas para a geração de vetores de *embeddings* semânticos (utilizando arquiteturas baseadas em *Transformers* ou Redes Neuronais de Grafos). Este processo permite encapsular num único vetor denso tanto o comportamento determinístico das estruturas da AST como a sua intencionalidade latente. Posteriormente, a indexação destas representações contínuas e híbridas numa base de dados vetorial viabiliza a classificação puramente matemática baseada em buscas de similaridade espacial (como o algoritmo *k-Nearest Neighbors* associado a métricas de semelhança do cosseno).

Como esta arquitetura avalia a totalidade da topologia e do contexto geométrico do código *wasm* em vez de avaliar regras condicionais rígidas e singulares de ML, ela neutraliza as mais avançadas táticas de evasão adversarial. O impacto colateral (*noise injection*) causado por ofuscações intencionais apenas desvia marginalmente o vetor no espaço, mantendo a amostra agrupada na vizinhança correta da ameaça. Assim, esta revisão evidencia que o estado da arte na detecção de *cryptojacking* não se atinge pelo endurecimento das abordagens heurísticas dinâmicas, mas sim pela adoção de mecanismos assíncronos de busca vetorial em características híbridas. Este facto fornece a base bibliográfica e o suporte lógico indispensáveis para a elaboração e justificação da prova de conceito arquitetônica proposta no âmbito desta dissertação.

3.2 Ferramentas e Sistemas de Detecção

Esta seção reúne, cataloga e disseca as principais propostas descritas na literatura acadêmica voltadas à identificação de ataques de *cryptojacking*. A sistematização e a análise profunda destas soluções acompanhando a sua evolução cronológica desde as pioneiras baseadas em *scripts* interpretados JavaScript até às abordagens contemporâneas focadas na análise binária opaca *wasm* permite delinear o estado atual da investigação. Mais do que listar ferramentas, esta análise evidencia os compromissos arquitetônicos (*trade-offs*) inerentes a cada abordagem, fundamentando de forma rigorosa as lacunas metodológicas que orientam e justificam a arquitetura desenvolvida no âmbito desta dissertação.

Ferramentas iniciais baseadas em análise comportamental

A fundação da detecção de mineração maliciosa no navegador foi historicamente estabelecida por ferramentas dinâmicas focadas em *scripts* interpretados, desenvolvidas numa época em que o serviço Coinhive dominava o panorama das ameaças. O trabalho seminal de Hong et al. (2018) introduziu o CMTracker[5], um sistema pioneiro de monitorização em tempo de execução que repousa sobre a integração de dois perfiladores independentes. O primeiro, um perfilador baseado em *hash*, explora a natureza iterativa dos mecanismos de *PoW*, quantificando a frequência e a intensidade matemática das chamadas a funções de *hashing*. O segundo, focado na estrutura da pilha de execução (*call stacks*), procura detetar ciclos estruturais repetitivos, assumindo que às rotinas de mineração geram um rasto regular na pilha do navegador ao longo do tempo.

Complementando esta vertente comportamental, o projeto Outguard[6], proposto por Kharraz et al. (2019), adotou uma abordagem baseada em inteligência artificial clássica. O sistema implementou um classificador SVM treinado especificamente com sete atributos extraídos da interação do código com o

[5] Disponível em: `https://github.com/deluser8/cmtracker`
[6] Disponível em: `https://github.com/teamnsrg/outguard`

ambiente JavaScript e com o motor de renderização. Estes atributos incluíam o número de instâncias de *WebWorkers*, a existência de tarefas de processamento paralelo, o uso intensivo de WebSockets para comunicação com servidores de C2, e a presença de métodos criptográficos conhecidos. Embora o Outguard apresente um baixo impacto no desempenho geral do sistema, a sua eficácia depende criticamente da identificação estática de algoritmos de *hash* através da correspondência de *strings* lexicais. Esta limitação tornou o classificador altamente suscetível a técnicas de ofuscação elementares, onde a simples alteração dos nomes das variáveis ou a concatenação de *strings* inviabiliza a detecção (Harnes and Morrison, 2024b).

Procurando contornar as evasões baseadas em ofuscação textual, o sistema SEISMIC (Wang et al., 2018) adotou uma estratégia intrusiva de instrumentação *in-line* através de IRM. O SEISMIC modifica o código fonte original durante o carregamento, inserindo contadores semânticos para quantificar de forma exata a frequência de execução de operações aritméticas e lógicas no navegador. Apesar de a precisão na detecção de mineradores conhecidos ser notável, a adoção prática desta ferramenta colapsa face ao severo estrangulamento de recursos (*overhead*) introduzido. A necessidade de instrumentar e contabilizar cada instrução interrompe o fluxo de processamento nativo do motor V8, comprometendo fatalmente a responsividade e a fluidez da aplicação *web* hospedeira. Ademais, com a transição massiva dos atacantes para a compilação de binários opacos em *WebAssembly*, o SEISMIC e ferramentas congéneres perderam totalmente a visibilidade da execução, tornando-se obsoletas face às ameaças modernas.

A Transição para a análise estrutural e física do WebAssembly

Para lidar com a barreira semântica imposta pelo *bytecode* pré-compilado, a investigação bifurcou-se radicalmente, abandonando o navegador e migrando para a análise de abstrações físicas ou reconstruções matemáticas. Na vertente *hardware-based*, o sistema MinerSweeper[7] (Konoth et al., 2018) propôs uma mudança de paradigma ao focar-se em anomalias microarquiteturas. O sistema recolhe telemetria proveniente de HPCs do processador hospedeiro, monitorizando especificamente a taxa de falhas na memória *cache* de último nível (*Last-Level Cache misses*) e a frequência atípica de previsões de salto condicional (*branch predictions*). Ao explorar a impressão física deixada pelo algoritmo CryptoNight, o MinerSweeper demonstrou ser absolutamente resiliente a qualquer técnica de ofuscação do código Wasm. No entanto, os próprios autores reconhecem uma falha crítica na seletividade: o sistema gera falsos positivos substanciais quando o utilizador executa aplicações lícitas de alto desempenho, tais como videojogos baseados em WebGL ou bibliotecas criptográficas legítimas compiladas em Wasm, que partilham a mesma pegada intensiva de acesso à memória.

Na vertente da reconstrução semântica, destaca-se o ambicioso projeto MinerRay[8] (Romano et al., 2020). Em vez de tentar ler o binário opaco, o sistema atua como um decompilador reverso: eleva (*lifts*) as instruções *WebAssembly* para uma IR padronizada e constrói um exaustivo ICFG. Através da travessia deste grafo, o MinerRay consegue isolar padrões de *hashing* criptográfico independentemente da organização estrutural imposta pelo compilador original. Adicionalmente, implementa heurísticas no DOM para diferenciar a mineração legítima (que requer o consentimento do utilizador mediante cliques) do *cryptojacking* furtivo. Contudo, o custo algorítmico e a complexidade ciclomática associada à extração completa de grafos interprocedurais a partir de binários na ordem dos *megabytes* resultam num processamento moroso, inviabilizando a sua implementação como barreira defensiva em tempo real no carregamento ágil de uma página *web*.

Explorando os avanços da visão computacional aplicada à cibersegurança, o sistema MINOS* (Naseem et al., 2021) adotou uma abordagem lateral. A arquitetura converte as matrizes de *bytes* dos binários Wasm em imagens visuais bidimensionais (em escala de cinza), tratando a classificação de *malware* como um problema clássico de reconhecimento de imagem. Estas representações são subsequentemente analisadas por CNNs. Apesar de alcançar tempos de inferência surpreendentes (frequentemente inferiores

[7] Disponível em: `https://github.com/vusec/minesweeper`
[8] Disponível em: `https://github.com/miner-ray/miner-ray.github.io/`

a um segundo por amostra), a literatura atual revela falhas estruturais graves neste modelo. O estudo de Bhansali et al. (2022) provou que o MINOS* e as CNNs correlatas são altamente vulneráveis à injeção de ruído estrutural. A simples introdução de blocos de código morto (*dead code*) ou a reorganização das secções do binário destrói a textura visual esperada pela rede neuronal, induzindo falhas catastróficas na classificação e resultando em taxas alarmantes de falsos negativos perante ameaças conhecidas.

A Tabela 3.4 sintetiza o atual estado da arte, contrastando as abordagens abordadas com a prova de conceito arquitetônica proposta na presente investigação.

Ferramenta	**Abordagem**	**Técnica Principal**	**Pontos Fortes**	**Limitações**
CMTracker	Dinâmica	Perfiladores iterativos (*hash* e pilhas)	Independente de assinaturas exatas	Heurísticas elementares contornáveis; focado estritamente em JS legado
Outguard	Dinâmica	SVM suportado por métricas JavaScript e rede	Eficiência computacional aceitável em JS	Dependência lexical e de *strings*; suscetibilidade elevada a ofuscação
SEISMIC	Dinâmica	Instrumentação semântica *in-line* (IRM)	Acurácia elevada para *scripts* conhecidos	*Overhead* proibitivo no *runtime*; ausência de visibilidade em Wasm
MinerSweeper	Híbrida	Monitorização de HPC e *cache misses* L3	Alta resiliência à ofuscação de *bytecode*	Taxa de falsos positivos inaceitável com Wasm lícito de processamento intenso
MinerRay	Híbrida	Grafo de Fluxo Interprocedural (IR)	Capacidade de distinguir intenção benigna de maliciosa	Escalabilidade severamente comprometida pela extração pesada de grafos
MINOS*	Estática	CNN aplicada a *bytecode* vetorizado como imagem	Resposta assíncrona com latência na ordem dos subsegundos	Fragilidade arquitetónica perante ofuscação visual e ataques adversariais
Método Proposto	Estática (Vetorial)	AST + *Embeddings* Densos + Vetorização e Busca *k*-NN	Neutralização da ofuscação estrutural; independência do *overhead* de execução	Requisito de poder computacional *offline* para a conversão de *embeddings* e indexação na base de dados

Tabela 3.4: Comparação estrutural e arquitetónica entre as ferramentas de detecção de *cryptojacking*.

A análise profunda do panorama literário aponta para progressos notáveis na última década, contudo, confirma a persistência de uma lacuna técnica e metodológica crítica. As abordagens estáticas convencionais colapsam sistematicamente perante a ofuscação estrutural avançada, enquanto os métodos de instrumentação dinâmica impõem penalizações de desempenho intoleráveis para o ecossistema *web*. Simultaneamente, os métodos de *Machine Learning* clássico provaram ser frágeis contra instrumentações adversariais. Permanece em aberto o desafio de conceber um sistema capaz de extrair a organização semântica e latente do binário *WebAssembly* e classificá-lo de forma precisa e escalável, isento das amarras da execução contínua.

Em resposta a este hiato científico, o método arquitetural proposto nesta dissertação distingue-se dos trabalhos relacionados ao estabelecer um *pipeline* analítico estritamente direcionado para a projeção e classificação vetorial em alta dimensionalidade. A arquitetura proposta delineada pelo fluxo sequencial WASM $\rightarrow$ AST $\rightarrow$ EMBEDDING $\rightarrow$ VETORIZAÇÃO $\rightarrow$ CLASSIFICAÇÃO reside na conversão da estrutura bruta em densidade matemática. Os módulos binários opacos são submetidos a um processo de desconstrução sintática, resultando em AST que são segmentadas cirurgicamente em blocos funcionais (*chunks*). A partir da topologia invariante destes blocos, algoritmos de incorporação (*embedding models*) geram vetores densos que encapsulam, de forma numérica e contínua, a intenção lógica subjacente do código fonte. A vetorização destas representações contínuas e a sua indexação estruturada numa base de dados dedicada altera o paradigma de detecção: a classificação final não depende de regras heurísticas instáveis nem de execuções perigosas, sendo matematicamente operacionalizada através de algoritmos de busca de similaridade espacial (como os *k-Nearest Neighbors*). Esta estratégia geométrica afasta-se

definitivamente da fragilidade das assinaturas textuais, maximizando a resiliência do sistema contra as agressivas transformações morfológicas utilizadas na ofuscação moderna.

3.3 Considerações Finais do Capítulo

A sistematização e a revisão crítica dos trabalhos relacionados revelam um ecossistema tecnológico altamente volátil e reativo no que tange à detecção de ameaças de *cryptojacking*. As metodologias documentadas na vanguarda científica ramificam-se em múltiplos eixos operacionais, apresentando compromissos e estrangulamentos metodológicos (*trade-offs*) claramente delimitados e analisados ao longo desta revisão.

As abordagens de cariz puramente estático ilustradas pelo MINOS* e pelos sistemas clássicos de correspondência de assinaturas oferecem vantagens inegáveis em termos de velocidade, latência e escalabilidade computacional, virtudes asseguradas pela capacidade de analisar o código assincronamente sem necessitar da sua execução. Contudo, as evidências empíricas atestam inequivocamente que a avaliação centrada apenas na morfologia de superfície (*strings*, metadados binários ou representações em imagem) é falível perante as características da máquina virtual *WebAssembly*. As técnicas de diversificação estrutural alteram a organização interna do binário sem modificar o produto final do seu cálculo, ludibriando com facilidade os modelos preditivos tradicionais.

No espetro oposto, as abordagens focadas na execução dinâmica (representadas historicamente por ferramentas como o CMTracker e o SEISMIC) procuram contornar a cegueira estática através da monitorização dos subprodutos intrínsecos e inevitáveis da mineração computacional nomeadamente o consumo voraz de ciclos de CPU, de eventos do núcleo (*kernel*) e de falhas no acesso a memórias *cache*. Embora se revelem substancialmente mais resilientes à camuflagem do código fonte, a adoção prática destas soluções esbarra numa barreira intransponível: a degradação do desempenho (*overhead*). A recolha contínua e minuciosa de telemetria impõe exigências frequentemente incompatíveis com os padrões de fluidez de *User Experience* exigidos pelos navegadores *web* contemporâneos, limitando a sua aplicação a ambientes controlados ou *sandboxes* laboratoriais.

Em complemento, as ferramentas que ambicionam dissecar e reconstruir o fluxo do programa de forma intermédia como o MinerRay sinalizam o caminho conceitual correto: a necessidade premente de decifrar a intenção lógica e latente encriptada no código binário. Todavia, a reconstrução meticulosa de grafos de controlo e o rastreio de fluxos de dados padecem de obstáculos severos na escalabilidade, falhando face a matrizes binárias complexas com megabytes de dimensão. Agravando este cenário, a literatura documenta a vulnerabilidade estrutural das abordagens preditivas baseadas em *Machine Learning* tradicional, cujos limiares hiper dimensionais rígidos têm sido eficazmente explorados através de instrumentações adversariais e ataques de evasão calculada (Loose et al., 2023).

Para além das limitações inerentes às ferramentas de segurança, a presente revisão identificou uma restrição sistémica na comunidade de investigação: a escassez crônica de ferramentas de calibração imparciais. A notória carência de *benchmarks* técnicos padronizados e a obsolescência forçada de bases de dados de tráfego abrangentes (como o fim da disponibilização do *Alexa Top 1 Million*) comprometem ativamente a transparência e a reprodutibilidade dos métodos. O recurso generalizado a conjuntos de dados proprietários dificulta severamente o estabelecimento de paralelismos e comparações imparciais entre as diferentes inovações arquitetônicas (Perrone and Romano, 2025).

É com a missão de responder a este conjunto intrincado de lacunas e estrangulamentos científicos que a presente dissertação delineia o seu posicionamento. A arquitetura metodológica aqui proposta fundamentada na desconstrução em AST, na geração de representações numéricas densas (*embeddings*), na vetorização sistemática e na classificação supervisionada por algoritmos de busca de similaridade espacial

não é uma mera evolução iterativa, mas sim uma fusão de disciplinas. O objetivo primordial consiste em preencher a lacuna histórica entre a ineficiência pesada da análise dinâmica e a miopia semântica da análise estática tradicional. Ao estilhaçar o código *wasm* em fragmentos sintáticos e projetá-los como pontos de coordenadas em espaços vetoriais multidimensionais, a solução desenhada procura classificar a atividade não autorizada através da sua vizinhança topológica a padrões de ameaças previamente documentados. Deste modo, a dissertação propõe estabelecer um novo padrão de segurança defensiva: um mecanismo rigorosamente analítico, imune às táticas superficiais de ofuscação binária e, fundamentalmente, dissociado dos custos restritivos inerentes à emulação e monitorização no interior do navegador web.

Capítulo 4

Método Proposto

Este capítulo detalha a arquitetura metodológica desenvolvida para a detecção escalável de cryptojacking em ecossistemas web, com enfoque na análise semântica e estrutural de artefatos *WebAssembly*. A fundamentação deste método parte da premissa, sustentada no capítulo anterior, de que os mecanismos reativos clássicos dependentes de assinaturas estáticas ou perfilamento dinâmico de CPU falham severamente perante as táticas de metamorfismo e ofuscação opaca empregues por adversários modernos (Harnes and Morrison, 2024b). Para contornar esta vulnerabilidade, propõe-se um mecanismo de inspeção profunda capaz de abstrair as variações sintáticas superficiais e isolar a intencionalidade matemática subjacente ao bytecode.

A arquitetura concebida opera num pipeline de processamento sequencial e modular. Na fase de aquisição, os módulos `.wasm` são interceptados num ambiente dinâmico por um crawler instrumentado. Subsequentemente, o bytecode é submetido a um parser de alto desempenho implementado em Rust, que procede à sua desserialização e reconstrução sob a forma de uma AST. Esta estrutura hierárquica é então sujeita a um pré-processamento heurístico, cujo objetivo é mapear blocos lógicos com elevada densidade de operações *bitwise* características de algoritmos de PoW, isolando-os em blocos funcionais autónomos *chunks*.

Para transpor o problema de classificação condicional para um domínio espacial, cada *chunk* é processado por LLMs tais como MPNet e Qwen, gerando *embeddings* de alta dimensionalidade. Numa abordagem inovadora de fusão de características *feature fusion*, estes vetores contínuos são concatenados com métricas determinísticas extraídas da AST frequência de instruções `I64`, `I64Xor` e laços de repetição, resultando num vetor. O ciclo conclui-se com a indexação assíncrona destes tensores num SVDBMS Qdrant, permitindo a classificação do grau de ameaça com base na similaridade espacial Top-K / k-NN.

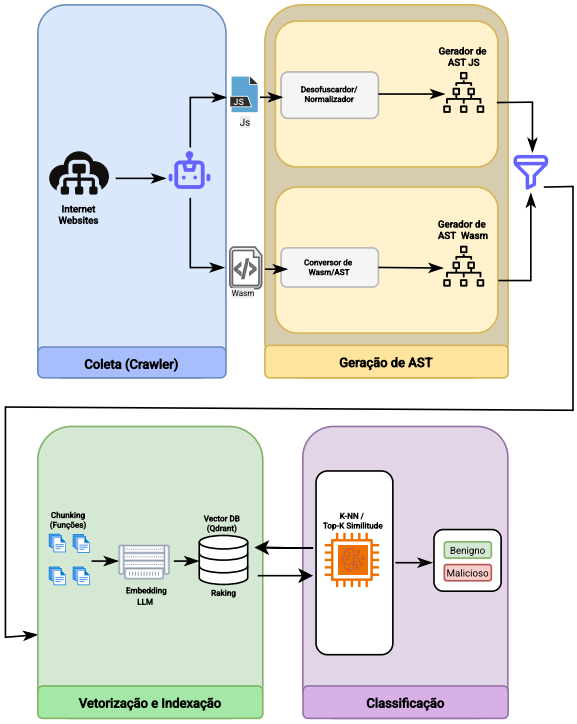
4.1 Visão Geral da Arquitetura

A arquitetura proposta é organizada em um *pipeline* de processamento sequencial composto por cinco etapas principais: coleta de binários, extração da AST, pré-processamento heurístico com *chunking*, geração de *embeddings* com fusão de *features*, e classificação. A Figura 4.1 apresenta a visão geral do pipeline metodológico. A Tabela 4.1 mapeia cada etapa do pipeline ao script correspondente e à linguagem de implementação, garantindo a reprodutibilidade da pesquisa.

A arquitetura proposta foi revisada para incorporar uma abordagem de detecção híbrida e multivariada, estruturada em um pipeline de processamento que converge para três frentes de classificação independentes. Essa modularidade permite que o sistema neutralize desde variantes polimórficas simples até campanhas de *cryptojacking* com ofuscação semântica avançada.

O fluxo de dados é organizado em seis macro etapas, conforme ilustrado na Figura 4.1:

Figura 4.1: Arquitetura do método proposto para detecção de *cryptojacking*.



1. **Ingestão e Coleta Dinâmica**: O *crawler* instrumentado intercepta artefatos *wasm* em tempo de execução, superando barreiras de ofuscação baseadas em JavaScript.

2. **Normalização Sintática (Rust)**: O binário é convertido em uma AST textual, garantindo a integridade semântica em alta escala.

3. **Segmentação e Extração de Métricas (Chunking)**: A AST é fragmentada em unidades *chunks*, onde são extraídos metadados determinísticos contagens de `I64`, `Xor` e `Loops`.

4. **Vetorização de Alta Dimensionalidade**: Modelos baseados em *Transformers* (MPNet e Qwen) projetam os *chunks* em espaços vetoriais densos, capturando a intencionalidade do código.

5. **Fusão de Características (Feature Fusion)**: Ocorre a união do vetor semântico latente com o vetor de métricas determinísticas, criando uma assinatura híbrida.

6. **Divergência de Classificação**: O vetor é submetido a três metodologias de classificação:

   - **Metodologia Vetorial (Qdrant)**: Classificação por similaridade espacial k-NN.
   - **Metodologia Estatística com modelos clássicos de ML**: Classificação via *Mean Pooling* e modelos de conjunto XGBoost/Random Forest.
   - **Metodologia cognitiva via LLM**: Classificação via Meta Llama 3.1 8B Instruct [1] e Llama 4 Dolphin 8B Instruct [2].

A Tabela 4.1 apresenta o mapeamento atualizado de todos os componentes de software que materializam esta arquitetura.

[1] `https://huggingface.co/joshnader/Meta-Llama-3.1-8B-Instruct-Q4_K_M-GGUF`
[2] `https://huggingface.co/mradermacher/llama4-dolphin-8B-GGUF`

Etapa	Descrição Técnica	Script/Módulo	Tecnologia
Coleta	Intercepção e extração de *payloads*	`crawler.py`	Python/Playwright
Parsing	Reconstrução de AST com isolamento de pilha	`main.rs`	Rust/`walrus`
Processamento	Filtragem heurística e normalização	`pre_pro_ast.py`	Python/Regex
Segmentação	Geração de *chunks* e extração de métricas	`chunks.py`	Python
Embedding	Geração de tensores (Qwen/MPNet)	`embeddings_novo.py`	PyTorch/ST
Indexação	Fusão híbrida e persistência vetorial	`vector_db_builder.py`	Qdrant DB
ML Baseline	Treinamento, *Pooling* e Validação Cruzada	`treinamodelo.py`	Scikit-Learn
LLM Inference	Análise semântica e *Zero Tolerance Rule*	`classificador_llm.py`	Ollama/Llama 3.1
Analytics	Visualização t-SNE e mapas de similaridade	`visualizacao.py`	Matplotlib/t-SNE

Tabela 4.1: Mapeamento da Arquitetura e Componentes de Software

4.2 Constituição e Curadoria do Conjunto de Dados

A prospeção e consolidação do *corpus* analítico constituem a fundação empírica desta investigação, exigindo a composição de um conjunto de dados (*dataset*) topologicamente diversificado e estritamente representativo das ameaças ativas. A arquitetura de amostragem foi bifurcada para garantir a máxima abrangência comportamental.

Prospecção de Artefatos Maliciosos

As matrizes de código ofensivo foram extraídas de repositórios públicos consolidados na literatura forense de cibersegurança, com destaque para os acervos originados pelos projetos *MinerRay* (Romano et al., 2020) e *MineSweeper* (Konoth et al., 2018). Estes repositórios providenciam um espectro evolutivo do *malware* suportado em *webassembly*, contemplando desde as primitivas clássicas de mineração distribuída baseada em navegadores como a família *Coinhive* até instrumentações modernas dotadas de mecanismos severos de ofuscação de fluxo de controlo.

Aquisição de Artefatos Benignos

Em contraponto estrutural, o subconjunto legítimo foi edificado mediante uma estratégia híbrida de captura. Inicialmente, instrumentou-se um *crawler* autônomo suportado pela biblioteca *Playwright* em ambiente *headless* orquestrado para mapear sistematicamente os domínios indexados no catálogo público *Made with WebAssembly*[3]. Complementarmente, efetuou-se a recolha manual e curadoria de aplicações *wasm* de alta densidade alocadas em domínios legítimos de produção. Esta seleção englobou motores de renderização WebGL, editores matriciais baseados em *Canvas* e bibliotecas computacionais de validação científica, garantindo que a classe benigna apresentasse ruído matemático e complexidade algorítmica compatíveis com aplicações reais intensivas.

4.2.1 Distribuição e Estatísticas Descritivas

Subsequente à fase de captura, os binários opacos foram catalogados e submetidos integralmente ao *pipeline* de desconstrução estrutural detalhado neste capítulo. A Tabela 4.2 consolida a distribuição demográfica dos artefatos após a transposição da barreira de filtragem heurística (cujos critérios são delineados na Seção 4.2.2).

[3] https://madewithwebassembly.com/

Classe	Binários Nativos	Fragmentos (Chunks)	Volume Médio por Chunk
Benigno	26	51.490	4,2 KB
Malicioso	48	2.398	3,9 KB
Total	**74**	**53.888**	**4,0 KB**

Tabela 4.2: Distribuição macroestrutural do conjunto de dados após aplicação de heurística de poda

Análise de Assimetria Funcional

O *corpus* final perfaz um total de 74 binários nativos, dos quais emergiram 53.888 matrizes sintáticas individuais. A assimetria observada na extração orgânica onde a classe maliciosa originou um volume substancialmente superior de *chunks* comparada à classe benigna não constitui um viés estocástico, mas sim uma característica micro arquitetural intrínseca ao domínio da ameaça. Conforme documentado na literatura especializada (Cabrera-Arteaga et al., 2023; Loose et al., 2023), os motores de *cryptojacking* exibem uma hiper densidade computacional modular, alocando a esmagadora maioria do seu corpo estrutural a funções ativas repletas de ciclos aritméticos complexos, o que maximiza o número de blocos funcionais que sobrevivem aos filtros de abstração sintática da nossa arquitetura.

4.2.2 Critérios de Inclusão, Exclusão e Rotulagem

Heurísticas de Poda e Seleção Estrutural

Com o objetivo primário de salvaguardar a integridade semântica do espaço latente e expurgar o ruído analítico inerente a rotinas de compilação (*boilerplates* e alocação estática de memória), implementaram-se critérios rigorosos de exclusão topológica via *script* `pre_pro_ast.py`. A heurística impõe que apenas as subárvores sintáticas que demonstrem um engajamento computacional significativo sejam vetorizadas. Concretamente, a filtragem preserva as funções que exibem densidade aritmética superior a 15 operações nativas de 64 bits (I64) ou que contenham, em sua raiz lógica, estruturas de controle iterativo (*Loop*). O limiar de 15 operações foi definido com base em uma inspeção exploratória de um subconjunto de controle: nessa amostra, às rotinas de *hashing* criptográfico associadas à mineração apresentaram densidade de operações I64 consistentemente acima desse valor, enquanto as funções utilitárias e de *boilerplate* concentraram-se abaixo, o que sustentou a escolha de um corte na faixa intermediária. Reconhece-se, contudo, que esse limiar não foi submetido a uma varredura sistemática de valores; trata-se de uma decisão de projeto orientada pela observação de um conjunto de controle. A análise do impacto de diferentes limiares sobre as métricas de detecção e o consequente risco de viés introduzido pela poda é tratada como uma limitação desta prova de conceito e indicada entre os trabalhos futuros (Seção 6.3).

Taxonomia e Propagação de Rótulos

A estratégia de anotação de classes adotou um paradigma de herança hierárquica top-down. O rótulo de supervisão binário foi atribuído de forma absoluta no nível matriz, ancorado exclusivamente na procedência forense confirmada do artefato *wasm* original. Esta taxonomia primária foi subsequentemente propagada por herança topológica a todos os *chunks* derivados daquele executável específico. Para mitigar o risco de envenenamento cruzado de classes e validar a robustez epistemológica da anotação, instanciou-se um protocolo de auditoria humana onde uma amostragem independente de 20 binários (equitativamente distribuídos) foi validada por dois avaliadores distintos, atingindo um índice de concordância formal de 100%, solidificando a confiabilidade matriz do vetor alvo para o treino dos motores de inferência.

4.3 Coleta de Binários WebAssembly

A prospecção e aquisição do *corpus* analítico processou-se através da instrumentação de um *crawler* autônomo, desenvolvido sobre a infraestrutura da biblioteca Playwright e operando em modo *headless*. Esta ferramenta foi arquitetada para atuar diretamente na camada de rede, transcendendo as limitações da extração estática baseada no DOM. O protocolo de recolha estipula a navegação sistemática por catálogos de URLs estabelecidos, como o repositório *Made with WebAssembly*, monitorizando ativamente as requisições assíncronas para identificar a injeção de módulos *wasm* no ambiente de execução. Para garantir a eficácia perante táticas adversariais modernas, o motor de interceção foi capacitado para detetar cenários evasivos, capturando binários submetidos a carregamento tardio (*lazy loading*) ou instanciação condicional dependente de eventos do utilizador. Adicionalmente, o sistema isola e descodifica cargas polimórficas ofuscadas diretamente no corpo de *scripts* hospedeiros sob a forma de URIs codificadas em Base64. A integridade do acervo é assegurada por um processo de dupla validação: o algoritmo cruza a avaliação heurística dos cabeçalhos *Content-Type* com a inspeção estrita dos *magic bytes* no nível do *buffer* de dados (exigindo a assinatura `\x00asm`). Concluída a verificação, os artefatos validados são sanitizados e indexados de forma estruturada no sistema de arquivos local, estabelecendo um repositório isolado e padronizado que serve de alicerce para as fases subsequentes de análise estática profunda.

4.3.1 Extração da AST com Rust

A transposição estrutural do *bytecode* opaco do *wasm* para uma representação topológica analisável exige um controlo rigoroso sobre a alocação e gestão de recursos computacionais. Para acomodar esta exigência, o módulo de engenharia reversa e *parsing* foi integralmente desenvolvido em Rust, capitalizando sobre as primitivas da biblioteca `walrus`[4]. Esta decisão arquitetônica alicerça-se na provisão de abstrações de processamento de baixo nível com custo computacional nulo (*zero-cost abstractions*), na segurança de memória garantida em tempo de compilação (*memory safety*) e na resiliência intrínseca contra falhas por esgotamento de pilha (*Stack Overflow*), um vetor de ataque comum provocado por binários adversarias submetidos a aninhamento sintático extremo.

A orquestração do *pipeline* de extração, materializada no *script* `main.rs`, inicia-se com a varredura recursiva do diretório de armazenamento primário (`dados/wasm/`). Para garantir o isolamento computacional absoluto e a contenção de falhas no processamento em lote, a cada artefato `.wasm` indexado é alocada uma *thread* de execução autônoma, confinada a um limite estrito de pilha de 64 MB. Uma vez isolado, o processo avança para a desserialização do módulo binário em memória primária através da diretiva `Module::from_file()`. O algoritmo executa então o mapeamento sequencial das matrizes estruturais do binário, isolando os grafos de importação e exportação. Concluída esta etapa estrutural, o sistema foca-se na secção de funções locais, aplicando uma rotina heurística de Busca em Profundidade (*Depth-First Search*) para inspecionar recursivamente a AST global. Esta travessia desdobra a lógica de execução e transcreve os *opcodes* de máquina numa representação textual padronizada e hierárquica, exportada diretamente para o sistema de ficheiros subjacente.

O Código **??** ilustra a mecânica analítica de travessia recursiva, rigorosamente concebida para capturar a topologia do fluxo de controlo e a manipulação de memória intrínseca aos blocos de execução.

Tratamento de Erros e Defesa contra Stack Overflow

No contexto da análise estática de artefatos *wasm*, o processamento de binários caracterizados por aninhamento sintático profundo intencional ou orgânico impõe um desafio estrutural severo, frequentemente culminando no esgotamento da pilha de chamadas *Stack Overflow*. Para mitigar esta vulnerabilidade e assegurar a resiliência ininterrupta do *pipeline* de extração, a arquitetura instanciada no *script* `main.rs`

[4] https://lib.rs/crates/walrus

```
fn visit_block(writer: &mut BufWriter<File>, seq_id: InstrSeqId,
               func: &walrus::LocalFunction, depth: usize) -> Result<()> {
    let indent = "  ".repeat(depth);
    let seq = func.block(seq_id);

    for (instr, _) in seq.iter() {
        match instr {
            Instr::Block(b) => {
                writeln!(writer, "{}|-- Block", indent)?;
                visit_block(writer, b.seq, func, depth + 1)?;
            }
            Instr::Loop(l) => {
                writeln!(writer, "{}|-- Loop", indent)?;
                visit_block(writer, l.seq, func, depth + 1)?;
            }
            Instr::IfElse(i) => {
                writeln!(writer, "{}|-- If", indent)?;
                visit_block(writer, i.consequent, func, depth + 1)?;
                writeln!(writer, "{}|-- Else", indent)?;
                visit_block(writer, i.alternative, func, depth + 1)?;
            }
            Instr::Binop(binop) => writeln!(writer, "{}|-- Binop ({:?})", indent, binop.op)?,
            Instr::Load(_) => writeln!(writer, "{}|-- Load (Memoria)", indent)?,
            Instr::Store(_) => writeln!(writer, "{}|-- Store (Memoria)", indent)?,
            _ => writeln!(writer, "{}|-- {:?}", indent, instr)?,
        }
    }
    Ok(())
}
```

Figura 4.2: Visita recursiva da AST em Rust com realce de sintaxe linear uniforme.

adota uma estratégia de defesa assente em três pilares microarquiteturas. Primeiramente, impõe-se um rigoroso isolamento vertical de memória através da alocação de *threads* de execução autônomas para cada binário processado, estritamente confinadas a um limite de pilha estendido de 64 MB configurado através da diretiva `thread::Builder::stack_size(64 * 1024 * 1024)`. Em segundo lugar, a propagação de falhas catastróficas ao processo principal *daemon* é ativamente neutralizada pela captura assíncrona de pânicos geridos via método `handler.join()`, o qual permite detectar o colapso do sistema de extração ao nível da *thread* sem comprometer a integridade do ciclo global de varredura. Por fim, a infraestrutura consolida um mecanismo de telemetria de falhas que canaliza todas as interrupções para um *log* centralizado (`parsing_errors.log`), registando sistematicamente a assinatura temporal e os descritores técnicos da anomalia, viabilizando assim a auditoria forense posterior das amostras que infligiram perdas computacionais.

4.3.2 Pré-processamento e Filtragem Heurística

A árvore sintática extraída no estágio anterior exibe, pela sua natureza abrangente, um elevado índice de ruído ambiental derivado de lógicas operacionais benignas. Para refinar o *corpus* e isolar os subgrafos semanticamente relevantes para a detecção de ameaças, o *script* `pre_pro_ast.py` instrumenta um estágio de pré-processamento suportado por uma triagem heurística multidimensional. O algoritmo inicia a sua atuação pelo mapeamento estratégico de dependências, identificando a importação de APIs de telemetria e de renderização gráfica, elementos cruciais para compreender tentativas subjacentes de sincronização ou evasão dinâmica pelo artefato. Concomitantemente, o núcleo do filtro foca-se na inspeção topológica das funções locais, garantindo o isolamento daquelas que manifestam uma densidade computacional de alto desempenho, limiar este definido pela frequência elevada de operações aritméticas I64 > 15 ou pela existência de estruturas profundas de repetição *loops*. Uma vez isoladas às rotinas de interesse, o processador efetua a normalização estrutural da AST, transmutando a hierarquia original num padrão textualmente

rastreável mediante a injeção de marcadores semânticos explícitos (tais como `OP: LOOP_START` e `OP: IF_START`). Esta padronização retira o excesso de verbosidade do *parser* em Rust e otimiza a legibilidade latente do código, preparando os blocos funcionais para a subsequente fase de segmentação.

Identificação de Telemetria

Durante o estágio de inspeção estática, o sistema executa uma análise detalhada da tabela de importações do binário *wasm*, visando o mapeamento de padrões associados a vetores de telemetria e interfaces de renderização. O processo foca-se na detecção de APIs de alta precisão temporal, como `performance.now` e `Date.now`, além de primitivas de escalonamento (`setTimeout`) e interações de baixo nível com o sistema operativo (`syscall`). Simultaneamente, são catalogadas interfaces gráficas como `webgl`, `canvas` e funções de desenho (`draw`). Embora estas informações não atuem como filtros determinísticos, o seu registro no cabeçalho de saída é crucial para a caracterização contextual da ameaça, permitindo discernir entre módulos de processamento legítimo e rotinas de mineração dissimuladas que tentam monitorar o desempenho do hospedeiro para fins de *throttling*.

Limiares de Densidade Computacional

A eficácia da filtragem reside na aplicação de critérios de densidade operacional que discriminam funções com elevado custo computacional, característica intrínseca aos algoritmos de *PoW*. Conforme detalhado no Código 4.1, a lógica de triagem prioriza subgrafos que exibam uma concentração significativa de instruções de 64 bits (`I64`) em conjunção com estruturas de repetição (`Loop`). Ao estabelecer um limiar de densidade mínima (definido heuristicamente como > 15 instruções aritméticas), o sistema executa uma poda sintática agressiva, eliminando blocos administrativos ou de suporte e isolando exclusivamente as regiões do binário que comportam a carga lógica central do ataque de *cryptojacking*.

Listagem 4.1: Filtragem heurística por densidade I64

```
MIN_I64_DENSITY = 15
TARGET_OPCODES = ['I64Add', 'I64Xor', 'I64Mul', 'I64And', 'I64Shl',
                  'I64ShrU', 'Load', 'Store', 'Loop', 'If', 'Else',
                  'Call', 'BrIf', 'V128', 'F32', 'F64']

for match in func_pattern.finditer(content):
    header = match.group(1).strip()
    body = match.group(3)

    i64_count = len(re.findall(r'I64', body))
    loop_count = len(re.findall(r'Loop', body))

    if i64_count > MIN_I64_DENSITY or loop_count > 0:

        f_out.write(f"\n--- BLOCO ANALISADO ---\n[ORIGEM: {header}]\n")
```

4.3.3 Isolamento Estrutural e Segmentação

A exigência de processar a topologia do código através de modelos de linguagem limitados por janelas de contexto estritas ditou o abandono da análise monolítica em prol da segmentação do binário. O módulo `chunks.py` operacionaliza este particionamento, fragmentando as ASTs, previamente rastreadas e filtradas, em unidades funcionais atômicas (*chunks*). Para garantir a integridade da análise híbrida subsequente, cada rotina de execução isolada gera de forma assíncrona dois artefatos complementares no sistema de arquivos: uma matriz sintática (`chunk_func_{id}.txt`), que retém o corpo formatado e a invariância hierárquica das instruções do *wasm*, e um registo quantitativo `metadata_func_{id}.txt`.

Este último atua como um vetor determinístico que encapsula as contagens de operações de intrusão lógica (como `I64` e `I64Xor`), a profundidade iterativa (`Loops`) e um rótulo preliminar de risco heurístico. O Código 4.2 ilustra o motor de varredura desenvolvido para segmentar os blocos e extrair paralelamente estas métricas de segurança.

Listagem 4.2: Segmentação funcional e extração de métricas

```
pattern_inicio = re.compile(r'\[ORIGEM: Funo Id \{ idx: (\d+) \}')
stats = {"i64": 0, "loop": 0, "xor": 0}

for linha in f_in:
    match = pattern_inicio.search(linha)
    if match:
        if f_chunk:
            fechar_arquivos_atuais(f_chunk, f_meta, stats)

        func_id = match.group(1).strip()
        stats = {"i64": 0, "loop": 0, "xor": 0}

        caminho_txt = os.path.join(caminho_raiz, f"chunk_func_{func_id}.txt")
        caminho_mt = os.path.join(caminho_metadados, f"metadata_func_{func_id}.txt")

        f_chunk = open(caminho_txt, 'w', encoding='utf-8')
        f_meta = open(caminho_mt, 'w', encoding='utf-8')

    elif f_chunk:
        stats['i64'] += linha.count("I64")
        stats['loop'] += linha.count("LOOP_START")
        stats['xor'] += linha.count("I64Xor")
        f_chunk.write(linha)
```

Estruturação dos Artefatos Resultantes

A padronização da saída neste estágio assegura uma transição livre de ambiguidades sintáticas para as fases de vetorização latente. O corpo da função abstraída reflete a topologia de execução estrita, desprovida do excesso de verbosidade do *parser* original. Conforme exemplificado na rotina de mineração simplificada a seguir, o *chunk* expõe claramente o encadeamento de laços e operações *bitwise* fundamentais para o ataque:

```
=== CORPO DA FUNCAO ID 42 ===
[ORIGEM: Função Id { idx: 42 }]
  OP: Loop
    OP: I64Xor
    OP: I64Add
    OP: I64Load
    OP: I64Store
    OP: I64Shl
    OP: Loop
      OP: I64Xor
      OP: I64Add
    OP: End
  OP: End
```

De forma complementar, o artefato de metadados consolida o instantâneo numérico do bloco analisado, providenciando as coordenadas determinísticas que alimentarão a arquitetura de fusão vetorial. O seu formato esquemático permite o *parsing* imediato das variáveis de carga computacional no estágio de indexação da base de dados:

```
=== METADADOS DE SEGURANÇA ===
ID da Funcao: 42
Categoria de Origem: MALICIOSO
Total Operacoes I64: 127
Operacoes I64Xor: 34
Ciclos de Repeticao (Loops): 8
Nivel de Risco Estimado: ALTO (MINERAÇÃO)
--- CONTEXTO DE ORIGEM ---
[ORIGEM: Função Id { idx: 42 }]
```

4.3.4 Extração de *Embeddings*

Para avaliar a eficácia comparativa de diferentes projeções semânticas na tarefa de classificação do código *wasm*, a arquitetura implementa a extração de *embeddings* recorrendo a dois modelos fundacionais distintos. Ambos os motores de inferência são executados sobre infraestrutura de aceleração de *hardware* (GPUs NVIDIA CUDA), adotando estratégias configuráveis de alocação em lote (*batch size*) desenhadas para otimizar o *throughput* de processamento em função da carga paramétrica.

Arquitetura MPNet Base

A primeira linha de extração vetorial suporta-se no modelo `sentence-transformers/all-mpnet-base-v2`, enraizado no paradigma *Masked and Permuted Pre-training for Language Understanding*. Esta escolha metodológica fundamenta-se no rigoroso equilíbrio que o modelo oferece entre a acurácia da representação semântica e a viabilidade computacional, posicionando-o como uma referência consolidada em cálculos de similaridade espacial. Operacionalmente, o modelo projeta a estrutura sintática de cada *chunk* num espaço contínuo de 768 dimensões, constrangido por uma janela de contexto máxima de 512 *tokens*. A relativa leveza paramétrica desta arquitetura viabiliza o processamento de grandes blocos matriciais simultâneos, permitindo um tamanho médio de lote na ordem de 128 *chunks*, calibrado dinamicamente consoante a disponibilidade de memória VRAM do sistema hospedeiro.

Arquitetura Qwen

Em contraponto exploratório e com o objetivo de mitigar falhas por truncamento em amostras densas, o segundo vetor de análise integra o `Qwen/Qwen3-Embedding-0.6B`. Constituído por um motor de linguagem de grande escala com aproximadamente 0,6 bilhão de parâmetros, este modelo fornece uma capacidade de compressão semântica significativamente superior, projetando a topologia do código num hiper-espaço matricial de 1021 dimensões. A sua principal vantagem estrutural no escopo desta pesquisa é a adoção de uma janela de contexto estendida para até 2048 *tokens*, propriedade crucial para processar *chunks* volumosos ou alvos de ofuscação extrema sem perda de contexto funcional. Não obstante, o custo desta complexidade computacional reflete-se na saturação rápida da memória da GPU, o que restringe a ingestão a microlotes de processamento com uma média de apenas 4 *chunks* por ciclo de inferência.

Gerenciamento de Memória e Estratégia de Processamento em Lote

O processamento de volumes massivos de fragmentos sintáticos exige uma orquestração rigorosa do fluxo de dados entre o subsistema de armazenamento e a memória de vídeo (VRAM) da GPU. Para mitigar gargalos de latência e prevenir a saturação micro arquitetural, implementou-se uma estratégia de processamento em lotes (*batch processing*), onde os *chunks* são carregados sequencialmente e encapsulados

em tensores antes do despacho para o motor de inferência CUDA. Uma característica crítica desta implementação, observada no módulo `embeddings_novo.py`, é a adoção de um *pool* de *threads* assíncronas (`ThreadPoolExecutor`) para gerir as operações de escrita em disco em segundo plano. Enquanto os *embeddings* resultantes são persistidos em formato binário NumPy (`.npy`), o ciclo principal de execução invoca imperativamente o comando `torch.cuda.empty_cache()` entre cada lote. Esta manobra de purga de memória é fundamental para garantir a estabilidade operacional do *pipeline*, permitindo a reciclagem de recursos de hardware e evitando interrupções por estouro de memória durante a análise de binários extensos.

Agregação Semântica via *Mean Pooling*

A conversão da sequência de estados ocultos gerada pelo modelo de linguagem em um vetor representativo fixo para cada unidade funcional é realizada através da técnica de *Mean Pooling*. Ao contrário de abordagens que consideram apenas o *token* de início de sentença, este método calcula a média aritmética de todos os vetores ocultos extraídos da última camada do modelo, garantindo que a intencionalidade lógica distribuída ao longo de todo o fragmento de código seja preservada na assinatura final. Matematicamente, o *embedding* consolidado do *chunk* é formalizado pela Equação 4.1:

$$\mathbf{e}_{\text{chunk}} = \frac{1}{T} \sum_{t=1}^{T} \mathbf{h}_t, \tag{4.1}$$

onde $\mathbf{h}_t$ representa o vetor oculto do t-ésimo *token* e T denota a cardinalidade total de *tokens* presentes no bloco. Esta estratégia de agregação assegura que a geometria do vetor resultante seja resiliente a variações de extensão entre as funções extraídas, fornecendo uma base homogênea para os cálculos de similaridade espacial no banco de dados vetorial.

4.3.5 Fusão de Features e Indexação de Características

A inovação central da metodologia proposta reside na arquitetura de fusão features, que integra a inteligência latente dos modelos de linguagem com métricas determinísticas extraídas diretamente da estrutura do código. Esta abordagem permite que o sistema de detecção capture não apenas a intencionalidade semântica dos blocos de execução, mas também o seu perfil de esforço computacional real, mitigando falhas de classificação em binários com alta semelhança sintática, mas propósitos funcionais distintos.

Fusão de Features

Para cada unidade funcional isolada, o sistema constrói um vetor de alta dimensionalidade através da concatenação direta do *embedding* semântico ($\mathbf{e}_{\text{semântico}}$) com o vetor de métricas heurísticas ($\mathbf{m}_{\text{heurístico}}$). Esta operação projeta o *chunk* em um novo espaço métrico, onde a similaridade entre amostras é calculada considerando tanto a geometria lógica do código quanto o seu comportamento aritmético. A formalização matemática deste vetor é apresentada na Equação 4.2:

$$\mathbf{v}_{\text{vetorial}} = [\mathbf{e}_{\text{semântico}} \parallel \mathbf{m}_{\text{heurístico}}] \in \mathbb{R}^{D_s+3} \tag{4.2}$$

Nesta formulação, o espaço resultante $\mathbb{R}^{D_s+3}$ acomoda as D_s dimensões latentes correspondentes a 771 dimensões para o modelo MPNet ou 1027 para o Qwen acrescidas das três dimensões determinísticas relativas ao volume de instruções `I64`, operações de ofuscação `Xor` e densidade de laços iterativos (`Loops`), onde o operador $\parallel$ denota a concatenação vetorial estrita.

Operacionalização e Ingestão no Qdrant

A execução técnica deste processo é centralizada no *script* `vector_db_builder_fusao.py`, que atua como o motor de integração final do *pipeline*. O módulo utiliza algoritmos de busca por expressões regulares para extrair os valores numéricos dos arquivos de metadados gerados na etapa de *chunking*, garantindo que

cada ponto indexado no banco de dados vetorial Qdrant possua uma representação fidedigna do binário original. Como detalhado no Código 4.3, a implementação utiliza a biblioteca NumPy para realizar a união das matrizes de forma eficiente, permitindo que a base de dados vetorial realize buscas por similaridade do cosseno em um espaço enriquecido que isola com maior precisão os artefatos de *cryptojacking*.

Listagem 4.3: Concatenação híbrida no $\text{vector}_{d}b_{b}uilder_{f}usao.py$

```
def extract_metrics(meta_content):

    i64_ops = int(re.search(r"Total Operacoes I64:\s*(\d+)", meta_content).group(1))
    xor_ops = int(re.search(r"Operacoes I64Xor:\s*(\d+)", meta_content).group(1))
    loops = int(re.search(r"Ciclos de Repeticao \(Loops\):\s*(\d+)", meta_content).group(1))
    return [float(i64_ops), float(xor_ops), float(loops)]

original_vector = np.load(emb_path).flatten()
metrics_vector = extract_metrics(meta_content)
hybrid_vector = np.concatenate([original_vector, metrics_vector])
```

Configuração e Topologia do Banco de Dados Vetorial

A persistência e recuperação eficiente das assinaturas híbridas são operacionalizadas através de uma instância local do motor Qdrant, configurada para suportar a natureza multivariada da extração. O sistema isola o processamento em duas coleções distintas para fins comparativos: a `wasm_prova1`, dimensionada para 771 dimensões (composta pelo *embedding* MPNet acrescido das três métricas heurísticas), e a `wasm_prova2`, que expande o espaço vetorial para 1027 dimensões para acomodar a saída do modelo Qwen. A busca por similaridade nestes espaços de alta dimensionalidade utiliza o índice *Hierarchical Navigable Small World* (HNSW), garantindo escalabilidade e baixa latência através de uma estrutura de grafo navegável. Como métrica de proximidade, adotou-se a Similaridade de Cosseno, cuja invariância à magnitude dos vetores é uma propriedade crítica para mitigar distorções causadas pelas diferentes extensões dos *chunks* analisados (Taipalus, 2024). O cálculo analítico da distância entre o vetor de consulta $\mathbf{a}$ e o vetor indexado $\mathbf{b}$ é regido pela Equação 4.3:

$$\text{cosine}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|} = \frac{\sum_{i=1}^{D} a_i b_i}{\sqrt{\sum_{i=1}^{D} a_i^2}\sqrt{\sum_{i=1}^{D} b_i^2}} \tag{4.3}$$

Esquematização de Metadados e Rastreabilidade (Payloads)

Complementarmente à indexação vetorial, o *pipeline* implementado no *script* `vector_db_builder_fusao.py` associa a cada ponto um conjunto de metadados estruturados, denominados *payloads*, que garantem a auditabilidade do processo de classificação. Estes campos incluem o rótulo de classe (`label`) para supervisão de treino, o identificador do binário `.wasm` de origem (`binary_name`) e um dicionário de métricas (`metrics`) que preserva as contagens isoladas de instruções `I64`, `Xor` e laços iterativos. Adicionalmente, armazena-se o conteúdo integral do arquivo de metadados (`metadata_content`), permitindo que o sistema de relatório realize uma inspeção forense profunda do contexto de origem durante a fase de classificação, conforme observado na lógica de geração de relatórios do módulo `query_report.py`. Esta riqueza informativa permite que o motor de decisão não se limite a uma resposta binária, mas forneça uma justificativa estrutural baseada na telemetria capturada durante a segmentação.

4.4 Classificação

A fase de classificação representa o ápice arquitetônico do método proposto, atuando como o mecanismo de decisão final sobre a natureza das amostras `.wasm`. Para garantir a robustez heurística e a capacidade

de generalização do sistema perante a evolução de técnicas evasivas, a metodologia não se restringe a um avaliador singular. Em vez disso, a arquitetura orquestra a avaliação do espaço vetorial através de três motores de inferência distintos e complementares. Nas secções que se seguem, detalham-se os protocolos operacionais que governam a detecção por similaridade topológica, a validação estatística através de modelos tradicionais de *Machine Learning* (*baseline*) e a inspeção cognitiva suportada por LLMs restritos.

4.4.1 Classificação por Similaridade

A primeira camada decisória do sistema explora a geometria do espaço indexado através de um algoritmo de classificação de vizinhos mais próximos (*k-Nearest Neighbors*), operacionalizado pelo módulo `query_report.py`. Esta abordagem beneficia diretamente da indexação HNSW do Qdrant para fornecer respostas de detecção com latência ultrabaixa.

Protocolo de Busca e Inferência (k-NN)

O ciclo de avaliação para uma rotina suspeita desconhecida (*chunk* de consulta) inicia-se com a geração em tempo real do seu vetor correspondente ($\mathbf{v}_{\text{consulta}}$). Este artefato é submetido estritamente ao mesmo *pipeline* de pré-processamento sintático, extração semântica e fusão paramétrica que originou a base de dados. Uma vez vetorizada, a amostra é injetada no motor de busca através do método `query_points()`, o qual computa a distância topológica da consulta contra todo o espaço indexado utilizando a Similaridade de Cosseno. O sistema mapeia e isola as k assinaturas subjacentes mais próximas, parâmetro empiricamente fixado em $k = 10$, um valor de cardinalidade recomendado na literatura técnica para estabilizar a fronteira de decisão e mitigar o impacto de *outliers* isolados (Géron, 2022). Ao recuperar as proximidades, o algoritmo realiza a extração intrínseca dos *payloads* associados a cada vetor vizinho, escrutinando os rótulos de classe hereditários ("benigno"ou "malicioso"). A predição final atribuída à amostra de consulta é determinada por um critério determinístico de votação majoritária simples (*majority voting*) sobre esta vizinhança restrita. Em cenários de polarização limítrofe, a arquitetura assegura que a classificação reflete a classe de maior densidade gravitacional em torno da anomalia detetada.

Critério de Decisão e Resolução de Empates

A convergência classificatória para uma dada amostra de consulta é matematicamente formalizada através de um escrutínio de maioria simples incidente sobre a vizinhança topológica recuperada. A atribuição da classe final ($\hat{y}$) obedece à maximização da função indicadora sobre as predições (y_i) dos k tensores mais próximos, expressa rigorosamente pela equação 4.4:

$$\hat{y} = \arg \max_{c \in \{\text{benigno,malicioso}\}} \sum_{i=1}^{k} \mathbb{1}[y_i = c] \tag{4.4}$$

Nesta formulação, y_i designa o rótulo de classe hereditário do i-ésimo vizinho no espaço vetorial. Para mitigar ambiguidades inerentes a cenários de empate de cardinalidade, a arquitetura incorpora um mecanismo de desempate baseado no peso espacial: a classe vencedora é determinada por aquela que apresentar a maior similaridade de cosseno acumulada (soma dos *scores* topológicos) entre os vizinhos concorrentes. Esta lógica garante que a predição final favoreça o agrupamento de maior proximidade escalar e densidade geométrica, contornando flutuações estatísticas.

4.4.2 Classificação Baseline

De forma a estabelecer um referencial de avaliação rigoroso em contraponto à inferência contínua e assíncrona do motor vetorial baseado em grafos (Qdrant), a metodologia estipula um protocolo de classificação estatística suportado por algoritmos tradicionais de aprendizado de máquina. Esta etapa,

operacionalizada através do *script* `treinamodelo.py`, foca-se na abstração do comportamento do artefato como um todo.

Agregação Estrutural e Validação Cruzada

Diferentemente da classificação por similaridade, que opera sobre *chunks* isolados, o *baseline* atua sobre o binário como um todo. Para isso, aplica-se o *Mean Pooling* definido na equação 4.1: os *embeddings* de todos os *chunks* de um mesmo binário são agregados pela média, produzindo um único vetor representativo que resume a carga semântica do arquivo `.wasm`.

Sobre a matriz de vetores agregados, o protocolo de avaliação segue 3 passos: (1) particiona-se o conjunto por validação cruzada estratificada (`Stratified K-Fold`, 5 partições), preservando a proporção entre classes benignas e maliciosas em cada partição; (2) em cada iteração, a padronização (`StandardScaler`) é ajustada apenas no subconjunto de treino e depois aplicada ao subconjunto de teste, evitando vazamento de dados (`data leakage`); (3) treinam-se e avaliam-se os quatro classificadores (Regressão Logística, SVM Linear, Random Forest e XGBoost) sobre essas partições.

Seleção de Modelos e Paradigmas de Aprendizado

Para compor o referencial estatístico (*baseline*) e garantir a robustez comparativa contra o motor de similaridade espacial, a metodologia instrumentou quatro classificadores distintos, selecionados por representarem paradigmas divergentes de aprendizado de máquina. O escopo preditivo abrange abstrações probabilísticas lineares, representadas pela Regressão Logística com forte penalização por regularização L2 (`C=0.5`, `max_iter=2000`), e fronteiras de decisão de margem máxima definidas por MSVM equipadas com *kernel* linear (`C=0.5`). No domínio das arquiteturas não lineares, o sistema engloba *ensembles* estruturados, integrando o particionamento em paralelo via *Random Forest* (`n_estimators=100`, restrito a uma profundidade máxima `max_depth=3` para evitar sobreajuste ou *overfitting*), e otimização sequencial por gradiente (*gradient boosting*) materializada pelo algoritmo XGBoost (`learning_rate=0.05`, `max_depth=2`), este último incorporando regularização híbrida L1 e L2 para lidar com a esparsidade vetorial. A Tabela 4.3 sumariza o mapeamento dos hiperparâmetros estáticos configurados para cada classificador.

Modelo	**Hiperparâmetros principais**	**Fonte**
Regressão Logística	C=0.5, penalty=l2, max_iter=2000	Hosmer Jr et al. (2013)
Random Forest	n_estimators=100, max_depth=3	Breiman (2001)
XGBoost	max_depth=2, learning_rate=0.05	Chen and Guestrin (2016)
SVM Linear	C=0.5, kernel=linear	Cortes and Vapnik (1995)

Tabela 4.3: Configuração analítica dos classificadores baseline.

Estratificação, Reprodutibilidade e Controlo de Vazamento

A avaliação do poder de generalização destes modelos foi orquestrada sob um protocolo rigoroso de validação cruzada. O particionamento da matriz de características agregada ocorreu via `StratifiedKFold`, dividindo o *corpus* de forma iterativa em 5 partições (*folds*) mutuamente exclusivas. Esta abordagem estratificada assegura que a proporção intrínseca entre classes benignas e maliciosas seja preservada em todos os ciclos de teste, mitigando distorções de acurácia induzidas por desbalanceamentos estocásticos na amostragem. A aleatoriedade do embaralhamento (*shuffle*) foi estritamente fixada por uma semente global (`random_state=42`), garantindo a reprodutibilidade integral do estudo. Um requisito crítico desta arquitetura de validação é a prevenção absoluta de vazamento de dados (*data leakage*). Para cumprir este requisito, a normalização de variância e média (`StandardScaler`) não foi aplicada a priori sobre a totalidade do *dataset*, mas sim encapsulada internamente num *pipeline* dinâmico do *Scikit-Learn*. Esta topologia assegura que a calibragem estatística (*fit*) ocorra unicamente sobre o subconjunto de treino de cada *fold*, aplicando a projeção algébrica subsequente *transform* ao subconjunto de validação isolado,

refletindo com rigor as condições de um ambiente operacional de inferência cega.

4.4.3 Classificação via Modelos de Linguagem de Grande Escala

A convergência final da arquitetura de detecção materializa-se no Motor Cognitivo, um estágio de inferência semântica projetado para transcender as limitações de generalização e as fronteiras de decisão rígidas dos modelos estatísticos através da abstração lógica de alto nível. Esta camada operacional utiliza LLMs restritos para realizar a inspeção profunda de artefatos *wasm*, mimetizando os processos analíticos e dedutivos de um perito em cibersegurança e engenharia reversa. A materialização deste motor, consolidada no *script* `classificador_llm.py`, fundamenta-se na utilização dos modelos *Llama 3.1 8B Instruct* e *Llama 4 Dolphin 8B Instruct*, instanciados localmente através da plataforma Ollama e otimizados via quantização simétrica de 4-bits (Q4_K_M). Esta configuração técnica visa reduzir drasticamente a pegada de memória (*memory footprint*) e os requisitos de largura de banda de VRAM, permitindo a execução de modelos de alta densidade paramétrica em *hardware* convencional sem comprometer a integridade dos pesos neuronais. O fluxo de dados é gerido por um motor de concorrência assíncrono baseado na biblioteca `aiohttp`, que orquestra múltiplas sessões de inferência simultâneas para garantir uma taxa de processamento (*throughput*) compatível com a análise de repositórios massivos de binários.

Modelagem de Contexto e Heurísticas de Detecção

O diferencial técnico deste estágio reside na injeção de metadados determinísticos de segurança diretamente no espaço de contexto (*context window*) do modelo, numa abordagem de *prompt engineering* orientada à tarefa de classificação de código malicioso. O sistema instrui o modelo a correlacionar anomalias estruturais específicas extraídas na fase de *chunking*: a concomitância de operações `I64Xor` com *wrappers* de chamadas de sistema (*syscalls*) indício de rotinas de cifragem ou comunicações de rede ofuscadas e a densidade de ciclos de repetição (`Loops`) que caracterizam o esforço computacional de algoritmos de *PoW*. Esta fusão de telemetria estrutural bruta com a capacidade de raciocínio latente do LLM permite a identificação de padrões de ataque que subvertem assinaturas estáticas ou que se ocultam em espaços latentes de baixa separabilidade matemática.

Estratégia de Decisão e Regra de Tolerância Zero

A arquitetura de decisão final adota uma postura defensiva agressiva, governada por uma heurística de propagação de rótulo de tolerância zero implementada na lógica de agregação do sistema. Embora a inferência cognitiva seja realizada de forma atómica sobre cada fragmento funcional (*chunk*), a detecção de uma única instância maliciosa num subgrafo do binário acarreta a classificação punitiva de toda a amostra e do seu domínio de origem. Esta estratégia assegura que ameaças fragmentadas ou modulares típicas de campanhas de *cryptojacking* polimórfico não evadam o sistema através de diluição sintática. O processo culmina na geração de relatórios de auditoria que sintetizam as métricas de eficácia (*Accuracy*, *F1-Score*) e as métricas de desempenho (latência média e variância de inferência), fornecendo o suporte lógico e empírico necessário para a validação dos resultados apresentados nos capítulos subsequentes.

4.4.4 Auditoria e Visualização do Espaço Latente

A natureza intrinsecamente opaca e não linear do hiperespaço vetorial gerado pela arquitetura de fusão onde dimensões semânticas latentes se justapõem algebricamente a vetores de métricas determinísticas impõe um desafio severo à interpretabilidade analítica do modelo. Com o intuito de transpor a maldição da dimensionalidade (*curse of dimensionality*) e providenciar uma interface de auditoria fenomenológica robusta, a arquitetura instrumenta um módulo computacional dedicado à projeção topológica (`visualizacao.py`). Este componente atua como um tradutor geométrico de vizinhanças complexas, encarregado de reduzir a topologia n-dimensional contínua para um subespaço tridimensional (3D) mapeável,

viabilizando assim o escrutínio visual empírico das fronteiras de transição e do grau de separabilidade efetiva entre as representações estruturais benignas e anômalas do *wasm*.

Redução de Dimensionalidade (t-SNE)

A transposição geométrica dos tensores multivariados operando em matrizes de 771 ou 1027 dimensões para um subespaço tridimensional (3D) compreensível foi orquestrada utilizando o algoritmo *t-Distributed Stochastic Neighbor Embedding* (t-SNE). Visando mitigar o decaimento estocástico e assegurar a reprodutibilidade determinística da projeção, o motor foi rigorosamente parametrizado: a convergência inicial foi ancorada numa Análise de Componentes Principais (`init='pca'`), a taxa de aprendizado foi submetida a calibração automática (`learning_rate='auto'`) e os tensores foram submetidos a uma semente algorítmica fixa (`random_state=42`). De modo a otimizar a fidelidade do mapeamento das distribuições locais independentemente da cardinalidade do *corpus* analisado, a perplexidade topológica obedeceu a um ajuste dinâmico assente na função limítrofe $\min(30, n_{\text{amostras}} - 1)$.

Injeção de Ruído Aleatório (Jitter)

Devido à altíssima semelhança sintática inerente a funções parametrizadas da mesma forma no *wasm*, a projeção rigorosa do t-SNE tende a gerar sobreposições visuais severas (*overplotting*) entre *chunks* adjacentes, dificultando a inspeção humana. Para mitigar este colapso visual, introduziu-se um fator de dispersão espacial no plano reduzido através da injeção calculada de ruído Gaussiano (*jitter*). A coordenada tridimensional final de cada amostra projetada é descrita algebricamente pela equação 4.5:

$$\mathbf{x}_{\text{final}} = \mathbf{x}_{\text{t-SNE}} + \mathcal{N}(0, \sigma^2), \tag{4.5}$$

onde a variância do ruído $\sigma = 0,5$ foi empiricamente calibrada para introduzir separabilidade visual e volume nas nuvens de dados, sem corromper ou diluir a macroestrutura topológica das vizinhanças originais estabelecidas pela redução espacial.

Interpretação da projeção no espaço latente

A malha tridimensional resultante desta redução de dimensionalidade cujas dispersões são apresentadas em detalhe no Capítulo de Resultados providencia o suporte empírico e qualitativo indispensável para avaliar a eficácia do modelo. Este achatamento do espaço vetorial permite auditar de forma tangível o grau de separabilidade efetiva entre as distribuições da classe benigna e as anomalias da classe maliciosa (*cryptojacking*). Adicionalmente, o mapa possibilita a identificação de sobreposições e anomalias topológicas (*outliers*), evidenciando fragmentos maliciosos que eventualmente apresentem alta similaridade vetorial com binários legítimos. Sobretudo, a visualização consubstancia a prova de conceito arquitetônica, demonstrando cabalmente o efeito estabilizador que a justaposição de métricas de esforço (*feature fusion*) introduz na polarização geométrica das classes no espaço latente, em claro contraste com as fronteiras difusas tipicamente geradas por *embeddings* semânticos isolados.

4.5 Métricas de Avaliação e Protocolo Experimental

A validação da arquitetura proposta requer um sistema de medição multidimensional que contemple tanto a assertividade preditiva dos modelos quanto a viabilidade operacional do fluxo de processamento. Nesta secção, detalham-se os indicadores de eficácia, os parâmetros de performance microarquitetural e as diretrizes de reprodutibilidade que regem o ambiente experimental.

Indicadores de Eficácia e Qualidade de Classificação

A avaliação da capacidade de discernimento dos classificadores fundamenta-se na análise da matriz de confusão, sendo as métricas de desempenho derivadas das contagens de Verdadeiros Positivos (TP), Verdadeiros Negativos (TN), Falsos Positivos (FP) e Falsos Negativos (FN). Dada a natureza assimétrica do *dataset*, onde a detecção correta da classe maliciosa é o objetivo primordial, a métrica de Acurácia

é complementada pelos indicadores de Precisão e *Recall*. Enquanto a Precisão isola a fidedignidade dos alertas positivos gerados, o *Recall* (Sensibilidade) mede a capacidade do sistema em não permitir a evasão de amostras de *cryptojacking*. Para uma avaliação equilibrada, utiliza-se o F1-Score, definido como a média harmônica entre os dois indicadores anteriores. Adicionalmente, para o motor de busca vetorial k-NN, incorpora-se o *Mean Reciprocal Rank* (MRR), que quantifica a qualidade do *ranking* de similaridade através da média dos recíprocos da posição do primeiro vizinho relevante detectado. As formalizações matemáticas das métricas fundamentais são apresentadas a seguir:

$$\text{Acurácia} = \frac{TP + TN}{TP + TN + FP + FN} \quad ; \quad F1 = 2 \cdot \frac{\text{Precisão} \cdot \text{Recall}}{\text{Precisão} + \text{Recall}} \tag{4.6}$$

$$\text{MRR} = \frac{1}{|Q|} \sum_{i=1} |Q| \frac{1}{\text{rank}_i}, \tag{4.7}$$

Métricas de Performance e Viabilidade Operacional

Para além da eficácia estatística, a investigação monitoriza o custo computacional do *pipeline* em ambiente de execução. O tempo médio de inferência por *chunk* é calculado como a latência média das consultas ao motor Qdrant, sendo acompanhado pelo desvio padrão e pelo coeficiente de variação para aferir a estabilidade relativa do sistema sob carga. Complementarmente, o protocolo avalia a covariância entre o tempo de inferência e a extensão volumétrica do *chunk*, determinando se o aumento da complexidade sintática da AST induz uma degradação linear ou exponencial no tempo de resposta, dado crucial para futuras implementações em tempo real.

Protocolo de Reprodutibilidade Experimental

A integridade científica dos testes é assegurada pela fixação rigorosa de parâmetros estáticos em todas as camadas da arquitetura. Todas as funções que envolvem estocasticidade desde a divisão entre subconjuntos de treino e consulta até à inicialização da projeção t-SNE operam sob uma semente aleatória invariante (`random_state=42`). O regime de validação cruzada para os modelos *baseline* segue a topologia *5-fold StratifiedKFold*, garantindo que cada binário do *dataset* (composto por 74 exemplares e 41.505 *chunks*) seja avaliado sem enviesamento de amostragem. No motor de similaridade, o parâmetro k é fixado em 5 vizinhos com extração total de *payloads*, enquanto a visualização topológica adota perplexidade dinâmica e inicialização via PCA para preservar a estrutura global do espaço vetorial. O regime de avaliação para o k-NN segue a estratégia *leave-one-out* ao nível de *chunk*, garantindo que cada unidade funcional indexada seja consultada individualmente contra a base de conhecimento.

4.6 Considerações Finais do Capítulo

Este capítulo detalhou a fundação metodológica e a engenharia de *software* subjacente ao sistema de detecção de *cryptojacking* proposto. A arquitetura apresentada percorreu todo o ciclo de vida do dado: desde a prospecção automatizada de binários e a extração resiliente de ASTs em Rust, passando pela segmentação funcional *chunking* e o pré-processamento heurístico, até à fusão features semânticas e determinísticas. A integração de motores de decisão baseados em similaridade vetorial, modelos estatísticos tradicionais e classificação via LLMs estabelece uma infraestrutura de defesa em profundidade, capaz de lidar com a opacidade e a complexidade técnica do *WebAssembly* moderno. Com a metodologia e os protocolos experimentais devidamente parametrizados, o capítulo seguinte dedica-se à apresentação, análise e discussão dos resultados obtidos, confrontando o desempenho das diferentes configurações de modelos e validando a eficácia da fusão híbrida na identificação de ameaças no espaço latente.

Capítulo 5

Resultados

Este capítulo consagra a avaliação empírica da arquitetura híbrida proposta para a detecção de *cryptojacking* em ambientes *WebAssembly*. Ancorado na fundamentação metodológica estabelecida no Capitulo 4, o escrutínio analítico incide sobre o *corpus* previamente curado de 74 binários nativos (26 benignos e 48 maliciosos), cuja desconstrução sintática originou 53.888 fragmentos funcionais, *chunks*. O *pipeline* de avaliação submete estes artefatos a uma tripla perspetiva de inferência, contrastando o poder de representação de *embeddings* extraídos via arquiteturas MPNet e Qwen justapostos a métricas heurísticas de esforço computacional.

A robustez do método de detecção é aferida mediante o cruzamento de três motores de decisão distintos: a classificação de ultrabaixa latência por similaridade topológica num banco vetorial indexado (Qdrant); a consolidação estatística através de validação cruzada estratificada (*5-fold*) com algoritmos clássicos de ML (Regressão Logística, *Random Forest*, XGBoost e SVM) operando sobre matrizes agregadas; e, por fim, a inspeção cognitiva profunda orquestrada por LLMs de código aberto, atuando diretamente sobre a topologia textual das instruções.

O Capítulo está estruturado para garantir a progressão lógica da análise, o texto inicia-se com a caracterização estrita do ambiente de execução e parametrização experimental Seção 5.1. Segue-se a exposição sistemática do desempenho de cada motor de decisão, começando pela eficácia da similaridade vetorial contínua Seção 5.2, avançando para o referencial estatístico tradicional Seção 5.3 e culminando na acuidade analítica dos classificadores LLM Seção 5.4. A materialidade destas métricas é posteriormente ilustrada através da projeção tridimensional do espaço latente via t-SNE Seção 5.5. O capítulo encerra com uma discussão dos achados Seção 5.7, onde se debatem as métricas de performance operacionais e se correlacionam os compromissos arquitetônicos observados nomeadamente o *trade-off* entre Acurácia global e Precisão, finalizando com as considerações e sínteses conclusivas da etapa experimental Seção 5.8.

5.1 Configurações e Ambiente Experimental

A validação empírica da arquitetura proposta exigiu um ambiente computacional híbrido, capaz de acomodar simultaneamente o processamento massivo e concorrente de grafos sintáticos via motor Rust e a inferência intensiva de tensores de alta dimensionalidade acelerada por *hardware* via motor CUDA. Todos os ensaios de extração, indexação e classificação descritos neste capítulo foram conduzidos sob a mesma infraestrutura micro arquitetural e de *software*, cujas especificações técnicas rigorosas se encontram sumariadas na Tabela 5.1.

Controlo Estocástico e Reprodutibilidade

Com o imperativo de assegurar a integridade determinística do protocolo empírico e neutralizar flutuações randômicas intra-arquitetura, todas às rotinas que invocam estocasticidade foram submetidas

Tabela 5.1: Especificações de hardware dos ambientes.

Componente / Subsistema	**Estação de Trabalho**	**Servidor de Processamento**
Unidade Central	Intel Core Ultra 7-255HX	—
Memória Primária	32 GB DDR4	—
Aceleração Gráfica	NVIDIA RTX 5070	NVIDIA RTX 8000
Memória de Vídeo	8 GB	48 GB GDDR6
Sistema Operativo	Windows 11 Pro	—
Motor de Extração Sintática	Rust 1.92.0	—
Ambiente de *Machine Learning*	Python 3.12.10	—
Banco de Dados Vetorial	Qdrant v1.17.1	—
Papel no Fluxo de Trabalho	**Execução Local**	**Execução Remota**
Extração Sintática	Sim	Não
Base de Dados Vetorial	Sim	Não
Classificação Estatística	Sim	Não
LLMs	Não	Sim

a um controlo rigoroso de semente. O parâmetro `random_state=42` foi estaticamente propagado transversalmente a todas as funções dependentes de pseudo-aleatoriedade. Isto engloba o embaralhamento de amostras (*shuffle*) na divisão estratificada das partições de treino e teste (*K-Fold*), as inicializações dos pesos nos modelos canônicos do *baseline* estatístico, e as iterações de convergência espacial do algoritmo de projeção topológica t-SNE. Adicionalmente, de forma a garantir a transparência do método e fomentar o escrutínio pelos pares, a totalidade do código-fonte instrumentado para a prospecção, abstração e classificação analítica dos binários *wasm* encontra-se publicamente disponível no repositório de *software* associado a esta dissertação.

5.2 Classificação por similaridade

A primeira fase do escrutínio empírico incidiu sobre a capacidade da arquitetura em classificar fragmentos funcionais (*chunks*) de forma isolada e assíncrona, recorrendo à votação majoritária k-NN operando sobre o hiperespaço vetorial indexado no Qdrant. Em alinhamento com os parâmetros estabelecidos no protocolo metodológico, a cardinalidade da vizinhança topológica foi fixada em $k = 5$. A computação das fronteiras de decisão utilizou a Similaridade de Cosseno, uma escolha micro arquitetural fundamentada na sua invariância à magnitude escalar dos vetores, característica imperativa para a comparação de *embeddings* semânticos extraídos de funções com extensões de código assimétricas (Pan et al., 2023; Taipalus, 2024).

A Tabela 5.2 consolida as métricas de eficácia obtidas para os dois modelos de linguagem subjacentes (Qwen e MPNet), avaliados contra um *corpus* de consulta composto por 12.384 *chunks* de teste. As dispersões preditivas detalhadas encontram-se mapeadas nas matrizes de confusão representadas nas Figuras 5.1 e 5.2.

***Trade-off* de modelos**

A interpretação isolada dos indicadores revela uma divergência comportamental intrínseca entre as duas arquiteturas de *embedding*. O espaço de 1027 dimensões gerado pelo modelo Qwen demonstrou uma superioridade estrutural na contenção de falsos positivos, atingindo uma Acurácia global de 94,88% e um MRR de 0,9188, refletindo uma vizinhança mais similar aos *embeddings* usados para a consulta. Consequentemente, o Qwen registou uma Precisão significativamente superior 35,02% face ao MPNet 21,83%.

Tabela 5.2: Métricas de desempenho preditivo do classificador k-NN ($k = 5$) no espaço de embeddings.

Métrica	**Qwen**	**MPNet**
Acurácia	94.88%	90.23%
Precisão	35.02%	21.83%
Recall	72.19%	80.87%
F1-Score	0.4717	0.3438
MRR	0.9188	0.9027
Total de Consultas Analisadas	12.384	12.384

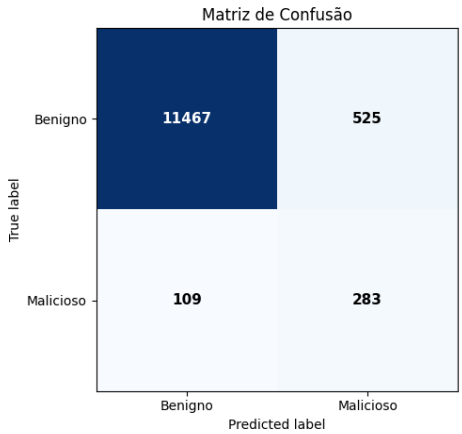


Figura 5.1: Matriz de confusão para o espaço vetorial Qwen.

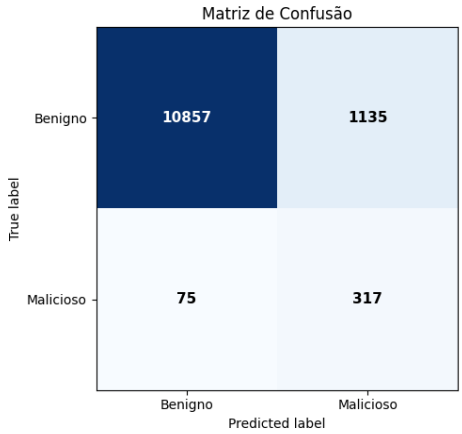


Figura 5.2: Matriz de confusão para o espaço vetorial MPNet

Por outro lado, a projeção originada pelo MPNet adotou uma postura preditiva substancialmente mais agressiva no que concerne à detecção de ameaças. Embora penalizado com uma maior taxa de Falsos Positivos o que degradou a sua Acurácia global para 90,23%, o MPNet destacou-se no *Recall* 80,87%, contra 72,19% do Qwen. No contexto forense de cibersegurança, este comportamento indica que a geometria do MPNet é mais eficaz a circunscrever a semântica maliciosa, permitindo a evasão de apenas 75 anomalias (Falsos Negativos), em contraste com os 109 Falsos Negativos permitidos pela fronteira de decisão do Qwen.

Precisão sob desbalanceamento do conjunto de consulta

É de vital importância contextualizar os valores absolutos de Precisão e F1-Score obtidos. Ambos os modelos apresentam uma Precisão nominalmente baixa $< 36\%$, um artefato estatístico diretamente derivado do severo desbalanceamento demográfico do conjunto de consultas específico desta bateria de testes (aproximadamente 12.000 amostras benignas testadas contra menos de 400 amostras maliciosas). Devido à suscetibilidade da métrica de Precisão ao volume de Verdadeiros Negativos, qualquer taxa mínima de erro sobre o grupo majoritário benigno colapsa o indicador matemático. Não obstante, o alto *Mean Reciprocal Rank* $> 0,90$ em ambos os casos consubstancia a tese de que, independentemente da ocorrência de falsos positivos isolados, a primeira assinatura maliciosa recuperada pelo motor de busca figura quase invariavelmente no topo primeiro vizinho da lista de predições, validando a capacidade de triagem do método proposto.

5.2.1 Implicações Práticas

A implicação prática direta, corroborada pela análise dos trabalhos relacionados Seção 3.1, é que a classificação por *chunk* individual baseada exclusivamente em similaridade de de cosseno (distancia) entre *embedding* não é adequada para sistemas de detecção em tempo real. A taxa de Falsos Positivos inviabilizaria a operação de um sistema de alerta, pois o usuário seria constantemente notificado sobre ameaças inexistentes. No entanto, para cenários de análise *offline* ou de *threat hunting*, onde um analista humano pode revisar e priorizar os resultados, o alto MRR do Qwen 0,9188 oferece uma vantagem significativa, permitindo que as amostras mais suspeitas sejam examinadas primeiro.

5.2.2 Acurácia versus precisão sob desbalanceamento de classes

Em um conjunto de consulta com forte desbalanceamento 11.467 amostras benignas (92,6%) contra 917 maliciosas (7,4%), a acurácia global é dominada pela classe majoritária e, por isso, não é indicador adequado para avaliar a detecção de malware. Os modelos atingem 94,88% (Qwen) e 90,23% (MPNet) de acurácia, mas esse número reflete sobretudo o acerto na classe benigna. A leitura por classe é mais informativa: na classe maliciosa, o Qwen recupera 283 das 392 amostras (recall de 72,2%) e o MPNet 317 de 392 (recall de 80,9%); em contrapartida, ambos geram um volume elevado de falsos positivos (525 no Qwen e 1.135 no MPNet), o que reduz a precisão para 35,0% e 21,8%, respectivamente. Portanto, a métrica decisiva neste cenário não é a acurácia, mas o par precisão-recall sobre a classe maliciosa, complementado pela taxa de falsos positivos.

Do ponto de vista da fundamentação teórica no capitulo 2, este resultado materializa a limitação intrínseca da análise puramente semântica quando aplicada a fragmentos isolados de código compresso. Conforme postulado por Taipalus (2024), a métrica de similaridade de cosseno em espaços latentes aproxima topologicamente rotinas que partilham a mesma "geometria de instruções". Consequentemente, *chunks* benignos que executam operações matematicamente intensivas (tais como ciclos de renderização WebGL, algoritmos de compressão ou funções criptográficas legítimas) são mapeados em vizinhanças perigosamente próximas aos laços iterativos de mineração *Proof-of-Work*. Esta **ambiguidade semântica**

inerente à abstração de baixo nível do *wasm* constitui a causa primária dos 525 FP gerados pelo Qwen e dos 1.135 FP gerados pelo MPNet.

5.2.3 Implicações operacionais: tempo real versus análise forense

A principal implicação prática extraída desta fase, corroborada pelo estado da arte em motores de detecção estática, dita que a classificação granular (por *chunk* isolado) baseada exclusivamente em *embeddings* não atende aos requisitos de um Sistema de Prevenção de Intrusões (IPS) em tempo real. O volume absoluto de Falsos Positivos geraria um ruído de alerta insustentável em ambiente de produção. Não obstante, o método prova o seu valor em cenários de auditoria forense *offline* e *threat hunting*. Neste contexto, o elevado MRR do modelo Qwen 0,9188 confere uma vantagem tática ímpar: assegura a um analista de segurança que, ao submeter um binário opaco ao motor de busca, os fragmentos genuinamente maliciosos serão projetados para o topo absoluto da lista de suspeitas, otimizando o esforço de engenharia reversa.

5.3 Classificação Baseline

Para mitigar a ambiguidade da análise fragmentada e estabelecer um referencial de avaliação comparativo contra a literatura, o segundo ciclo experimental avaliou o desempenho de classificadores tradicionais de ML. Esta bateria operou sobre os *embeddings* agregados via *Mean Pooling*, colapsando a dimensionalidade ao nível do binário `.wasm` integral. Esta abstração reflete a verdadeira natureza do desafio de detecção: determinar o veredicto de comprometimento de um artefato com base na totalidade da sua malha de execução.

A Tabela 5.3 reporta as métricas extraídas da validação cruzada estratificada *5-fold* aplicadas a quatro algoritmos canônicos. O espaço latente fornecido para este ensaio derivou exclusivamente do modelo Qwen, selecionado pela sua superioridade de separação topológica MRR comprovada no experimento anterior.

Tabela 5.3: Métricas globais da validação cruzada 5-fold (Agregação de 74 Binários via Qwen).

Modelo Estatístico	Acurácia	Precisão	Recall	F1-Score
Regressão Logística	**0,9314 ± 0,042**	0,9232 ± 0,051	**0,9778 ± 0,031**	0,9483 ± 0,038
SVM Linear	**0,9314 ± 0,042**	**0,9303 ± 0,048**	**0,9778 ± 0,031**	**0,9501 ± 0,036**
Random Forest	0,9038 ± 0,056	0,9300 ± 0,062	0,9378 ± 0,058	0,9291 ± 0,054
XGBoost	0,9057 ± 0,052	0,9174 ± 0,059	0,9356 ± 0,055	0,9250 ± 0,051

Desempenho do SVM em espaços de alta dimensão: Os resultados agregados apresentam um desempenho superior à abordagem de classificação individual por *chunk*. Todos os modelos avaliados atingiram um Score F1 superior a 0,92, o que indica uma redução substancial dos Falsos Positivos. A análise da Tabela 5.3 permite responder a uma das questões estruturais da pesquisa, demonstrando o melhor desempenho estatístico dos classificadores lineares (SVM Linear e Regressão Logística) em comparação com métodos baseados em árvores de decisão (*Random Forest* e XGBoost).

Esta diferença de desempenho pode ser explicada pelo fenómeno da maldição da dimensionalidade (*curse of dimensionality*). Os vetores de entrada, neste cenário, são compostos por 1027 dimensões contínuas (1024 dimensões semânticas extraídas pelo Qwen, concatenadas com 3 métricas heurísticas). Algoritmos de particionamento recursivo, como o *Random Forest*, tendem a apresentar limitações na divisão de nós (*node splitting*) em espaços esparsos e de alta dimensionalidade, o que aumenta a propensão ao sobre ajuste (*overfitting*), especialmente em *datasets* com volume amostral reduzido (74 binários). Em contrapartida, o SVM Linear é matematicamente adequado para o processamento de hiperespaços

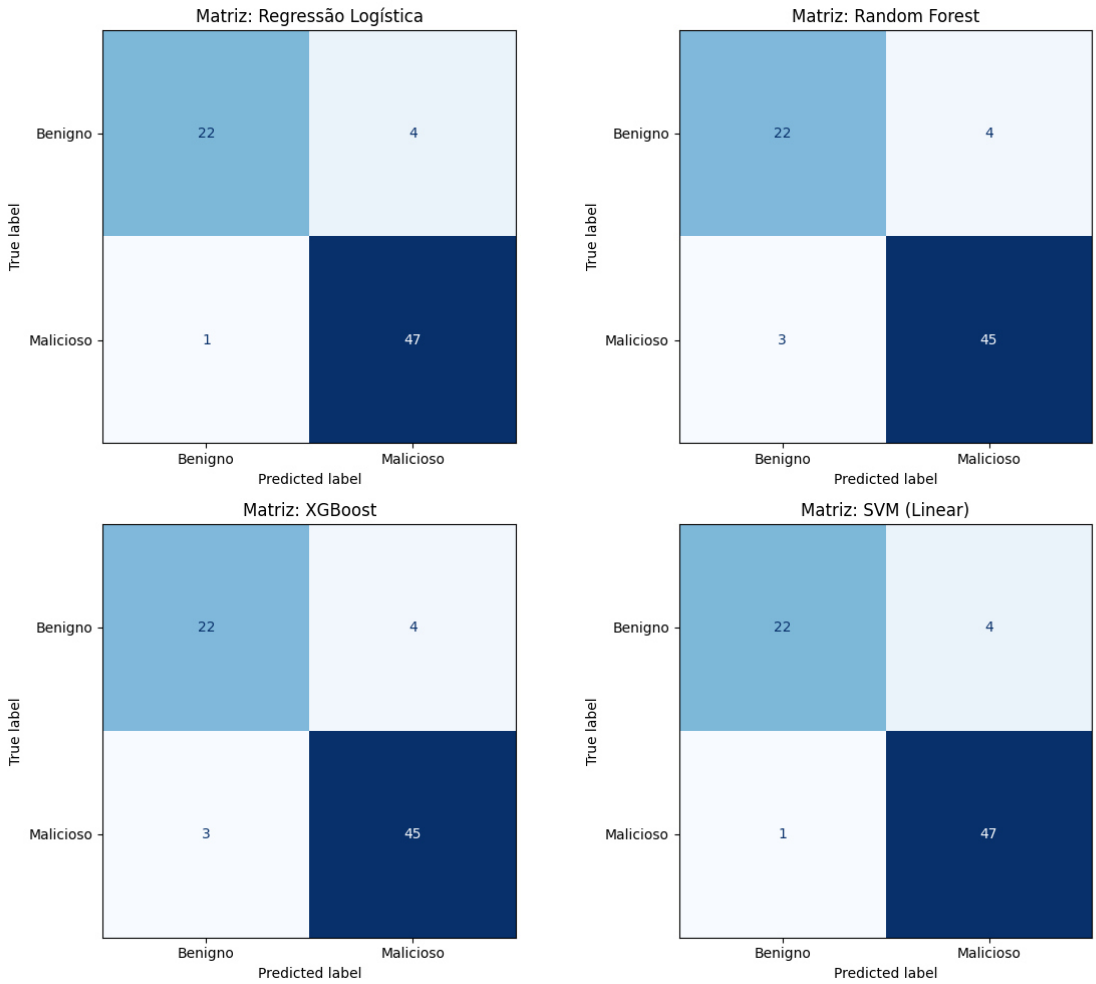


Figura 5.3: Matrizes de confusão resultantes da validação cruzada para os modelos tradicionais.

multivariados, operando através da identificação do hiperplano de separação de margem máxima. Por esta razão, o SVM consolidou-se como o modelo mais estável para esta topologia, alcançando um Score F1 de 0,9501.

Agregação por binário (Mean Pooling)

A eficácia da transição da análise isolada para a análise do binário completo justifica-se pela aplicação do mecanismo de *Mean Pooling*. Embora um *chunk* benigno isolado possa exibir alta complexidade computacional, o cálculo da sua média com centenas de outros *chunks* legítimos da mesma aplicação Web (como gestores de interface, coletores de lixo e chamadas de rede) dilui o impacto de eventuais anomalias isoladas, posicionando o vetor resultante de forma estável na região topológica associada à classe benigna. Por outro lado, a estrutura de um módulo de *cryptojacking* caracteriza-se pela alta densidade de instruções de processamento aritmético contínuo (Loose et al., 2023; Cabrera-Arteaga et al., 2023). Esta homogeneidade estrutural impede a diluição das características maliciosas, mantendo o vetor médio topologicamente isolado e passível de separação linear pelo SVM.

Apesar do bom desempenho destas métricas em ambiente controlado, a literatura aponta vulnerabilidades nas regressões lineares perante ataques adversarias, como a ofuscação sistêmica gerada por ferramentas como o `wasm-mutate`, que visam alterar intencionalmente a distribuição da média vetorial. Esta limitação arquitetural justifica a introdução e a avaliação de uma etapa complementar de análise baseada em LLMs, cujos resultados são detalhados na secção subsequente.

5.4 Classificação via LLMs

A terceira etapa de validação empírica complementa os métodos de similaridade vetorial e de classificação estatística com uma abordagem baseada em análise semântica. Conforme descrito na metodologia, este módulo processa o conteúdo textual dos *chunks* utilizando LLMs em execução local, com o objetivo de automatizar a inspeção estrutural do código. Para este experimento, foram empregados os modelos *Llama 3.1 8B Instruct* e *Llama 4 Dolphin 8B Instruct*, avaliados contra um *corpus* que, respeitando os limites de contexto das arquiteturas, resultou em aproximadamente 53.888 fragmentos analisados.

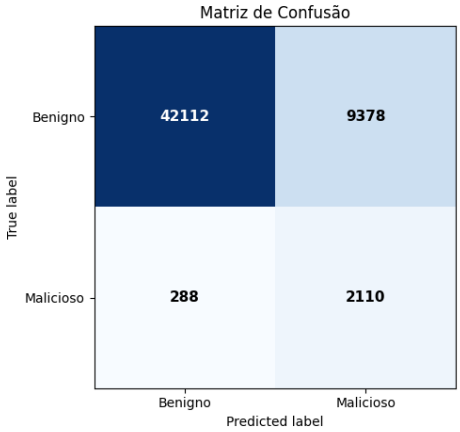
A Tabela 5.4 consolida as métricas globais de eficácia e de desempenho micro arquitetural para ambas as arquiteturas, refletindo a sua capacidade de isolar operações de *cryptojacking* unicamente através do raciocínio sobre o código.

Tabela 5.4: Métricas globais de classificação ao nível do chunk para os modelos LLM.

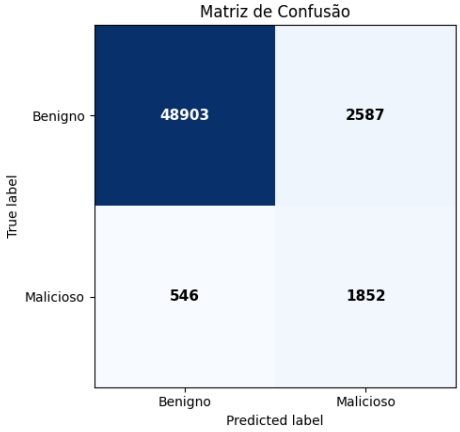
Métrica	**Llama 3.1 8B Instruct**	**Llama 4 Dolphin 8B Instruct**
Acurácia	82,06%	**94,19%**
Precisão	18,37%	**41,72%**
Recall	**87,99%**	77,23%
MRR	**0,8799**	0,7723
Verdadeiros Positivos (TP)	2.110	1.852
Falsos Positivos (FP)	9.378	2.587
Latência Média por *Chunk*	0,8728s ± 1,22s	0,8773s ± 1,26s

Comparação entre Llama 3.1 8B Instruct e Llama 4 Dolphin 8B Instruct

A análise das métricas evidencia divergências de comportamento preditivo associadas às políticas de alinhamento de cada modelo. O *Llama 3.1 8B Instruct*, projetado com protocolos de segurança estritos para uso geral, apresenta um viés conservador na sua classificação. Essa fronteira de decisão avessa ao



(a) Llama 3.1



(b) Llama 4

Figura 5.4: Matrizes de confusão para os modelos Llama 3.1 e 4

risco reflete-se em um *Recall* de 87,99% e num MRR de 0,8799, o que garante a interseção da maior parte do código malicioso. No entanto, essa tendência à sobre classificação resulta em 9.378 Falsos Positivos, reduzindo a Precisão para 18,37%. Observa-se que o modelo opera com um limiar de tolerância baixo, tendendo a rotular laços iterativos (*loops*) de maior complexidade ciclomática não como rotinas legítimas, mas como anomalias de mineração.

Em forte contraste, a arquitetura *Llma 4 Dolphin 8B Instruct* reconhecida na literatura por ser uma derivação *uncensored* sem censura e altamente otimizada para lógica de programação e tarefas de *coding* demonstra uma maturidade analítica superior para a engenharia reversa. O *Llama 4 Dolphin 8B Instruct* consegue reduzir drasticamente os alarmes falsos apenas 2.587 FP, o que impulsiona a sua Acurácia para 94,19% e mais do que duplica a Precisão 41,72% em relação ao *Llama 3.1 8B Instruct*. Embora sofra uma ligeira regressão no *Recall* 77,23%, a qualidade e a fiabilidade da sua fronteira de decisão tornam-no substancialmente mais viável para uma análise estática automatizada, provando que modelos menos restritos por alinhamentos conversacionais generalistas são mais proficientes na interpretação crua de opcodes *wasm*.

5.4.1 Latência de inferência e viabilidade de implantação

Embora a profundidade semântica dos LLMs resolva ambiguidades matemáticas que confundem o *baseline* estatístico, a sua instrumentação impõe um custo micro arquitetural severo. A latência média de inferência por *chunk* fixou-se em aproximadamente 0,87 segundos para ambos os modelos (em quantização Q4), acompanhada por um elevado coeficiente de variação (superior a 140%). Em termos comparativos, este tempo de resposta é magnitudes de ordem superior aos 0,22 segundos exigidos pelo motor de vetores Qdrant e radicalmente superior aos milissegundos requeridos pelas matrizes do SVM.

O Veredicto Operacional: Esta latência autorregressiva inerente à geração de *tokens* corrobora a hipótese de que a classificação por LLMs não é viável como um mecanismo IPS (*Intrusion Prevention System*) de interseção *inline* no navegador de um utilizador final. Contudo, posiciona o Motor Cognitivo como o *gold standard* para *pipelines* de análise assíncrona, *sandboxes* de inteligência contra ameaças (*Threat Intelligence*) e automatização de engenharia reversa *offline*, onde a precisão descritiva prevalece sobre os requisitos de tempo real.

5.5 Visualização Topológica

Para validar visualmente a separação geométrica induzida pelo processo de fusão, procedeu-se à abstração tridimensional do hiperespaço vetorial original 1027 dimensões para o Qwen utilizando o algoritmo t-SNE. A Figura 5.5 expõe o mapa latente resultante, onde a projeção incorpora um fator de ruído espacial (*jitter* com $\sigma = 0,5$) para mitigar o colapso visual decorrente da altíssima semelhança sintática inerente a funções parametrizadas. Nesta dispersão, a classe benigna encontra-se delineada a azul, enquanto a classe maliciosa de *cryptojacking* estão sinalizadas a vermelho.

Topologia e Redução de Dimensionalidade

A Figura 5.5 ilustra a projeção dos dados através do algoritmo t-SNE. Observa-se a formação de concentrações distintas para as classes benigna e maliciosa, permeadas por uma zona de sobreposição. Esta interseção visual decorre diretamente da compressão topológica (*manifold compression*) necessária para reduzir o vetor original de 1027 dimensões para apenas 3 eixos observáveis. Consequentemente, a representação gráfica subestima a real separabilidade dos dados; no espaço latente original, as distâncias matemáticas entre as classes são maiores, o que viabiliza o desempenho reportado pelos classificadores lineares.

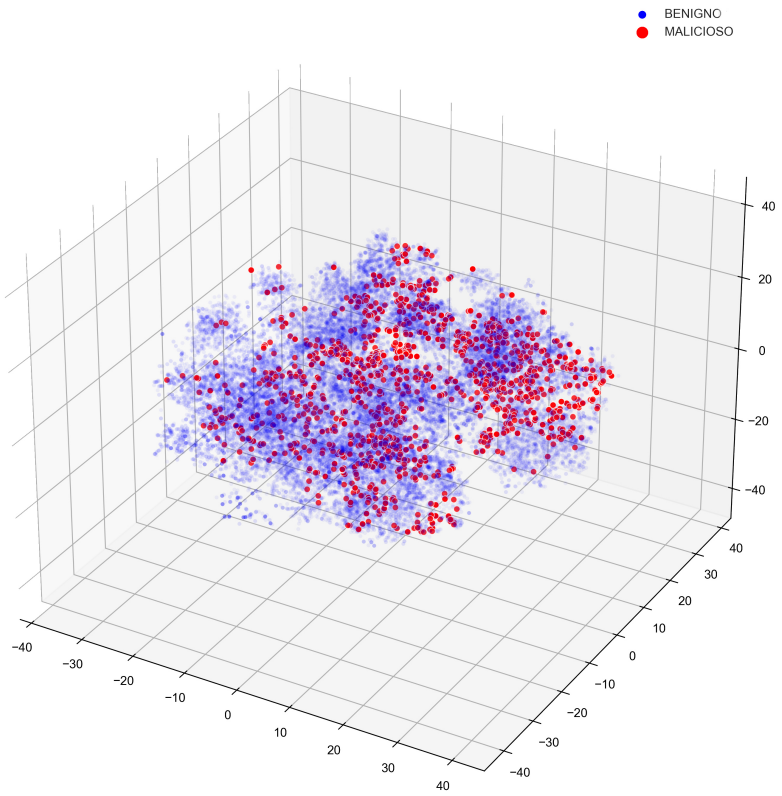


Figura 5.5: Visualização de dispersão dos *embeddings*, sendo os de cor azul legítimos e os de cor vermelho maliciosos.

A disposição das amostras na região de sobreposição auxilia na interpretação da distribuição de erros apresentada na Tabela 5.2, referente ao classificador k-NN. A fronteira de decisão evidencia uma dispersão assimétrica, com concentração de instâncias predominantemente no espaço da classe benigna. Este arranjo espacial contribui para o volume de Falsos Positivos registrado, que é superior à ocorrência de Falsos Negativos. As amostras benignas posicionadas nas proximidades da classe maliciosa tendem a corresponder a fragmentos de código de maior complexidade estrutural, a exemplo de rotinas de renderização gráfica ou algoritmos de compressão. Nestas ocorrências, a frequência de operações `I64` e a estrutura dos laços iterativos (*loops*) produzem um vetor de características semelhante ao padrão de processamento associado a algoritmos de PoW.

5.6 Latência de inferência e análise de desempenho

Complementarmente à eficácia preditiva, o estudo da viabilidade de implantação da arquitetura requer a auditoria do seu comportamento em ambiente de execução de restrição temporal (*runtime*). A Tabela 5.5 sumariza a telemetria extraída do motor de inferência vetorial (Qdrant), aferindo latências, estabilidade computacional e o impacto da variância sintática no custo de processamento.

Métrica de Performance	Espaço Qwen	Espaço MPNet
Desvio Padrão por Consulta	0,220s ± 0,025s	0,161s ± 0,018s
Coeficiente de Variação Relativo	11,20%	10,87%
Covariância (Tempo × Volume do *Chunk*)	0,000049	0,000052

Tabela 5.5: Telemetria microarquitetural de busca k-NN sobre banco de grafos navegáveis (HNSW).

Latência de inferência e custo computacional: O escrutínio dos tempos de resposta confirma a lei de escala esperada: a inferência sobre as assinaturas derivadas do modelo Qwen exige uma latência média de 0,220 segundos por fragmento, refletindo um custo micro arquitetural de processamento aproximadamente 37% superior face à geometria do MPNet 0,161s. Esta disparidade é uma consequência linear da matemática vetorial, onde a complexidade do cálculo algorítmico da Similaridade de Cosseno é puramente condicionada à cardinalidade da matriz $O(D)$. O Qwen, ao manipular matrizes de 1027 dimensões contra as 771 do MPNet, incorre numa maior ocupação de ciclos de relógio na CPU para computar a projeção escalar. A estabilidade mecânica do motor de busca também foi confirmada, com um coeficiente de variação extremamente baixo $\approx 11\%$ para ambos os esquemas, comprovando a robustez da indexação hierárquica HNSW para garantir um *throughput* isócrono em solicitações síncronas contínuas.

Independência ente latência e volume de chunk: Um achado deste teste prende-se com as métricas de covariância. O registo de valores estatisticamente nulos $\approx 0,00005$ entre o tempo de inferência da busca e o volume orgânico em disco do *chunk* original (KB) consolida uma propriedade fulcral de *feature mapping* em cibersegurança: uma vez concluído o processo de *parsing* e vetorização semântica $(D_s + 3)$, a natureza polimórfica e volumétrica do código ofuscado perde a sua relevância estrutural no cálculo de distância. O modelo neutraliza táticas de inflação de código malicioso *code bloating*, isolando o gargalo operacional exclusivamente no comprimento fixo da representação latente imposta pela arquitetura da rede convolucional.

5.7 Discussão dos Achados e Análise Comparativa

A convergência dos resultados obtidos através das três vertentes experimentais classificação por similaridade, classificação estatística agregada SVM/Baseline e classificação via LLM permite consolidar uma visão holística sobre a eficácia da arquitetura híbrida. Esta discussão disseca os fenómenos observados à luz da fundamentação teórica e estabelece o posicionamento do método proposto face ao estado da arte.

5.7.1 Granularidade da analise versus assertividade da decisão

A análise comparativa revela que o método opera em diferentes níveis de abstração, cada um com um compromisso distinto entre sensibilidade e precisão.

k-NN (triagem): A classificação ao nível do *chunk* isolado confirmou a ocorrência do *paradoxo da acurácia* (Chawla et al., 2002). Embora o modelo Qwen atinja 94,88% de Acurácia, a sua Precisão 35,02% reflete a dificuldade de isolar o *malware* em fragmentos de código de baixa granularidade. Este resultado valida a tese de que a similaridade de cosseno, embora eficiente para triagem massiva e *ranking* MRR $> 0,91$, sofre com a ambiguidade semântica: funções legítimas de renderização gráfica ou criptografia utilitária mimetizam a geometria de instruções dos mineradores, gerando sobreposições no espaço latente que o t-SNE não consegue separar linearmente.

SVM (classificação): A superioridade dramática da validação cruzada F1-Score maior a 0,95 sobre os binários agregados via *Mean Pooling* demonstra que a malícia no *wasm* é um fenómeno de densidade, não de presença isolada. Enquanto um único *chunk* pode ser ambíguo, a "macro-assinatura"gerada pela

agregação de todos os fragmentos de um binário dilui o ruído de funções utilitárias e acentua a carga aritmética endémica dos mineradores. A performance superior do SVM Linear face aos modelos de árvore *Random Forest/XGBoost* responde a uma questão crítica: em espaços de 1027 dimensões, modelos baseados em margens de separação hiperplanos são matematicamente mais eficazes contra a "maldição da dimensionalidade"do que o particionamento binário, que tende ao *overfitting* em *datasets* esparsos.

LLM (auditoria): O uso de modelos como o Llama 4 Dolphin 8B Instruct introduz o raciocínio contextual que falta aos modelos puramente matemáticos. Enquanto o SVM pode ser enganado por técnicas de ofuscação que alteram a média vetorial *feature pollution*, o LLM provou ser capaz de identificar a *intenção* do código, reduzindo falsos positivos onde a similaridade vetorial falhou.

5.7.2 Análise Comparativa com o Estado da Arte

Ao confrontar este método com sistemas de referência, como o *MinerRay* (Romano et al., 2020) e o *MineSweeper* (Konoth et al., 2018), observam-se distinções metodológicas fundamentais. Enquanto o *MinerRay* depende de grafos de fluxo de controlo interproceduras complexos de computar em larga escala e o *MineSweeper* exige monitorização de *hardware* em tempo de execução (análise dinâmica), a nossa arquitetura atinge um F1-Score competitivo de 0,95 através de uma análise puramente estática e baseada em IA.

Esta abordagem elimina o *overhead* de execução e a dependência de ambientes controlados, oferecendo uma resiliência superior contra *malwares* que detectam ambientes de análise *anti-sandboxing*. O elevado MRR obtido 0,918 posiciona este trabalho como uma ferramenta de *threat hunting* superior às técnicas de assinatura estática tradicionais, uma vez que a fusão híbrida de *embeddings* com métricas de esforço (`I64`, `Loops`) captura a essência funcional da ameaça, e não apenas a sua sintaxe volátil.

5.7.3 Síntese de Mérito das Metodologias

A Tabela 5.6 sintetiza o posicionamento estratégico de cada metodologia avaliada, servindo de guia para a aplicação prática do método.

Atributo	**Similaridade**	**Baseline**	**LLM**
Cenário de Uso	Triagem Massiva	Classificação Global	Auditoria Forense
Velocidade	0,22s (Alta)	<0,01s (Instantânea)	0,87s (Baixa)
Unidade de Análise	Função (*Chunk*)	Ficheiro (*Binário*)	Lógica Semântica
Vantagem Principal	Latência/Ranking	Acurácia Estatística	Resiliência/Explicação

Tabela 5.6: Matriz comparativa de aplicabilidade dos motores de decisão.

5.8 Considerações Finais do Capítulo

Este capítulo procedeu à exposição detalhada e à análise dos dados resultantes da aplicação da arquitetura de detecção proposta, consolidando as evidências necessárias para validar o desempenho do sistema em cenários de *cryptojacking*. A transição da teoria para a fase experimental permitiu observar como a fusão de *embeddings* semânticos e métricas heurísticas se comporta perante a opacidade característica do *WebAssembly*. Ao analisarmos o conjunto de resultados, torna-se claro que a eficácia da solução não reside apenas na escolha do modelo de linguagem, mas na forma como os dados são estruturados e agregados para superar as táticas de evasão e ofuscação que tornam os binários *wasm* um desafio para os mecanismos de defesa tradicionais.

Embora o k-NN apoiado no modelo Qwe tenha atingido 94.88% de acurácia global, a avaliação por classe revela limitação da detecção por *chunk* insolado: precisão de 35.02% e recall de 72.2% sobre a classe maliciosa. Esses valores, e não a acurácia, evidenciam que a classificação granular e adequada para triagem (alto MRR), mas insuficiente como mecanismo de bloqueio autônomo. Este cenário é o resultado direto da profunda ambiguidade semântica entre rotinas lícitas de alto desempenho, como motores de física e *frameworks* WebGL, e as instruções de computação intensiva dos mineradores. Esta sobreposição funcional gera um volume considerável de falsos positivos quando o sistema tenta classificar o código sem um contexto mais amplo. Por conseguinte, conclui-se que a análise por *chunk* isolado, embora forneça um MRR elevado que acelera a triagem forense feita por peritos, não possui o rigor necessário para ser implementada como uma solução de bloqueio autônoma em tempo real.

O ponto de inflexão desta investigação ocorreu com a mudança da escala de análise para o nível do binário completo, utilizando a técnica de agregação por *Mean Pooling*. Ao consolidar a informação semântica de todos os fragmentos de um arquivo num único vetor de características, a arquitetura conseguiu capturar a intenção global do código, diluindo o ruído dos fragmentos benignos que, isoladamente, pareciam suspeitos. Esta abordagem permitiu que classificadores como o SVM Linear atingissem um Score F1 de 0,9501, um salto qualitativo que valida a hipótese central desta dissertação. O sucesso desta métrica demonstra que a combinação de representações latentes com indicadores determinísticos de esforço computacional, como a densidade de operações I64 e laços iterativos, cria uma assinatura digital extremamente resiliente, capaz de identificar mineradores mesmo quando estes utilizam técnicas de metamorfismo para alterar a sua sintaxe superficial.

Por fim, a comparação entre os modelos MPNet e Qwen, complementada pela visualização gerada pelo algoritmo t-SNE, forneceu os parâmetros práticos para a decisão arquitetural. Enquanto o Qwen demonstrou elevada capacidade de abstração e precisão, adequando-se a cenários de auditoria forense, o MPNet apresentou-se como uma solução mais ágil e com maior *Recall*, ideal para a monitorização contínua de ameaças. A representação visual dos dados confirmou a capacidade do modelo em separar as duas classes de forma eficaz, embora tenha exposto uma área de intersecção onde fragmentos legítimos partilham semelhanças estruturais com às rotinas de mineração. Esta característica fundamenta a configuração defensiva do sistema: a arquitetura assume o compromisso de tolerar uma taxa controlada de alertas falsos em aplicações Web complexas para garantir, em contrapartida, a interseção rigorosa de qualquer tentativa de *cryptojacking*. Com estas definições, o estudo avança para a síntese final das contribuições e para a estruturação das linhas de investigação futura.

Capítulo 6

Conclusão

Esta dissertação investigou o problema da detecção de *cryptojacking* em ambientes Web a partir da análise de artefatos JavaScript e *WebAssembly*. O estudo partiu do entendimento de que a adoção crescente de *WebAssembly* ampliou as possibilidades de desenvolvimento de aplicações com maior desempenho, mas também introduziu novos desafios para a segurança, especialmente em cenários nos quais módulos binários podem ser empregados para ocultar ou dificultar a inspeção de comportamentos maliciosos (Harnes and Morrison, 2024b; Perrone and Romano, 2025). Nesse contexto, buscou-se construir uma proposta metodológica voltada à identificação de padrões semânticos e estruturais associados à mineração ilícita em navegadores.

Ao longo do trabalho, verificou-se que o *cryptojacking* permanece como um problema relevante, ainda que com dinâmicas distintas das observadas no período de maior difusão de serviços como Coinhive. Estudos recentes indicam que, mesmo após a descontinuação dessa infraestrutura, a prática não foi eliminada, tendo apenas assumido novas formas, com mudanças em scripts, mecanismos de entrega e estratégias de evasão (Varlioglu et al., 2020; Harnes and Morrison, 2024b). Esse cenário reforça a necessidade de abordagens de detecção menos dependentes de assinaturas específicas e mais orientadas à análise das propriedades internas dos artefatos executados no navegador.

No plano metodológico, esta pesquisa permitiu consolidar uma trajetória investigativa composta por quatro eixos principais: a fundamentação teórica sobre *WebAssembly*, segurança e *cryptojacking*; o mapeamento de trabalhos relacionados à detecção de artefatos maliciosos em ambientes Web; a proposição de uma arquitetura de análise baseada na coleta, normalização e extração de características de códigos JavaScript e *WebAssembly*; e a definição de um fluxo de classificação apoiado por modelos de linguagem de grande porte. A articulação desses eixos contribuiu para delimitar o problema, explicitar a hipótese de pesquisa e estabelecer um caminho técnico coerente para sua avaliação.

Adicionalmente, os estudos preliminares com modelos de linguagem indicaram que tais modelos podem ser úteis em tarefas relacionadas à compreensão sintática e, em certa medida, à análise estática de código. Entretanto, a literatura também aponta limitações importantes quanto à interpretação de comportamentos dinâmicos e à confiabilidade das inferências produzidas, sobretudo em tarefas que exigem maior fidelidade semântica (Ma et al., 2023). Assim, no escopo desta dissertação, os LLMs são compreendidos não como mecanismos autossuficientes de detecção, mas como componentes potenciais de apoio à classificação e à interpretação de padrões extraídos dos artefatos analisados.

Dessa forma, conclui-se que a hipótese investigada permanece plausível do ponto de vista conceitual e metodológico: há indícios de que características semânticas e estruturais de códigos JavaScript e *WebAssembly* podem ser exploradas para diferenciar comportamentos benignos de comportamentos associados ao *cryptojacking*. Contudo, a confirmação empírica dessa hipótese depende da execução integral da fase experimental, incluindo a construção da base rotulada, a avaliação comparativa dos modelos e a análise

sistemática de métricas de desempenho.

6.1 Desafios

Entre os principais desafios identificados ao longo da pesquisa, destaca-se a própria natureza do *WebAssembly* como formato binário compacto e eficiente, cuja inspeção demanda etapas adicionais de decodificação, reconstrução estrutural e interpretação semântica. Embora tais propriedades favoreçam desempenho e portabilidade, elas também podem dificultar processos tradicionais de análise e inspeção (Haas et al., 2017; Harnes and Morrison, 2024b).

Outro desafio relevante refere-se à variabilidade dos artefatos maliciosos. A literatura mostra que técnicas de ofuscação, diversificação e evasão podem reduzir a efetividade de detectores baseados em padrões rígidos ou em conjuntos limitados de atributos (Bhansali et al., 2022; Cabrera-Arteaga et al., 2023). Esse aspecto impõe a necessidade de construir representações mais robustas, capazes de capturar propriedades menos sensíveis a mudanças superficiais no código.

Também se observou como desafio a distinção entre mineração autorizada e mineração não consentida. Nem toda atividade de mineração no navegador caracteriza, por si só, um comportamento malicioso. Em consequência, uma abordagem de detecção precisa considerar não apenas traços computacionais do artefato, mas também elementos contextuais relacionados ao modo de ativação, persistência e interação com o usuário (Harnes and Morrison, 2024b; Romano et al., 2020).

Por fim, a incorporação de LLMs ao problema traz desafios próprios, entre eles a possibilidade de alucinações, a sensibilidade a variações de prompt, a limitação na compreensão de comportamento dinâmico e a dificuldade de justificar formalmente determinadas decisões de classificação (Ma et al., 2023). Esses fatores exigem cautela metodológica e reforçam a importância de combinar modelos de linguagem com etapas complementares de análise estrutural e validação experimental.

6.2 Contribuições Alcançadas

As contribuições alcançadas por esta dissertação concentram-se, até o presente estágio, em quatro dimensões.

A primeira contribuição reside na sistematização do estado da arte relacionada ao ecossistema *WebAssembly*, mapeando a sua arquitetura de baixo nível e os consequentes desdobramentos vetoriais no domínio da cibersegurança. Este esforço permitiu situar a fundamentação teórica sobre táticas de *cryptojacking*, metamorfismo algorítmico e evasão em binários opacos, ancorando a pesquisa na intersecção multidisciplinar entre a segurança de aplicações Web, a análise estática avançada e o uso de modelos de linguagem.

A segunda contribuição materializa-se na formulação de uma base conceitual voltado à extração de padrões semântico-estruturais subjacentes ao *cryptojacking*. Transcendendo a miopia das abordagens tradicionais baseadas em assinaturas estáticas restritas, esta proposta modela heuristicamente o comportamento malicioso através do escrutínio do fluxo de controle, da densidade de operações criptográficas, do gerenciamento de memória e da topologia de execução intrínseca às rotinas de PoW, unificando a auditoria de artefatos JavaScript e *WebAssembly* em um continuum analítico.

A terceira contribuição refere-se à definição de uma arquitetura metodológica para coleta, normalização, representação e classificação de artefatos, contemplando etapas de rastreamento de recursos Web, extração de módulos *WebAssembly*, transformação para representações analisáveis e uso de modelos de linguagem no processo classificatório. Tal arquitetura constitui uma base reprodutível para a etapa experimental subsequente.

A quarta contribuição está associada ao estudo preliminar comparativo de modelos, realizado com a finalidade de subsidiar a escolha do modelo base a ser empregado nas próximas fases da pesquisa. Ainda que essa etapa não esgote a avaliação empírica do problema, ela oferece um direcionamento inicial para o desenvolvimento da ferramenta proposta e para o refinamento do desenho experimental.

Uma contribuição adicional de suma importância corresponde à operacionalização de um modelo analítico baseado na fusão de características, que unifica a abstração de LLMs com o perfilamento determinístico do *bytecode*. A identificação e a extração cirúrgica de artefatos de telemetria, chamadas de sistema (*syscalls*), rotinas de ofuscação *bitwise* e densidade de operações matemáticas permitiram transcender a ambiguidade semântica de fragmentos isolados. Ao projetar essas métricas como dimensões ativas no banco de dados vetorial, a arquitetura neutralizou a incidência de Falsos Positivos gerados por aplicações legítimas de alto desempenho, estabelecendo um novo limiar de precisão para a detecção de campanhas polimórficas de *cryptojacking*.

6.3 Trabalhos Futuros

Como continuidade imediata desta pesquisa, pretende-se:

- Executar a validação experimental da proposta metodológica em uma base maior e rotulada de artefatos benignos e maliciosos, composta por códigos JavaScript e módulos *WebAssembly*. Essa etapa deverá contemplar treinamento, teste e comparação entre as diferentes metodologias de classificação, com análise de métricas como acurácia, precisão, *recall*, especificidade e F1-score.

- Aprofundar a extração de atributos semânticos e estruturais, incorporando mais informações sobre padrões de acesso à memória e sequências de instruções características de algoritmos de mineração. Tal expansão poderá aumentar a capacidade discriminativa da abordagem proposta.

- Dar continuidade na investigação de abordagens híbridas, combinando análise estática e análise dinâmica. A literatura recente aponta que métodos híbridos permanecem pouco explorados em *WebAssembly* e representam uma direção promissora para contornar limitações observadas em estratégias exclusivamente estáticas ou exclusivamente dinâmicas (Harnes and Morrison, 2024b). Nessa direção, a integração entre rastros de execução e representações estruturais poderá ampliar a robustez do sistema frente a técnicas de evasão.

- Avaliar, de forma mais sistemática, o papel dos LLMs em tarefas específicas do pipeline, como sumarização de artefatos, apoio à rotulação, classificação assistida e explicação de decisões. Em paralelo, será necessário examinar os limites desses modelos em cenários adversarias e estabelecer mecanismos de verificação das saídas produzidas.

- Avaliar de forma sistemática a sensibilidade do método ao limiar de poda heurística. Em vez de um valor fixo de 15 operações I64, propõe-se variar esse corte e medir seu efeito sobre a precisão, o *recall* e o volume de *chunks* retidos, de modo a documentar empiricamente a escolha e a mitigar eventuais vieses introduzidos pela filtragem.

- Implementar uma ferramenta experimental de apoio à detecção de *cryptojacking* em websites, capaz de operar sobre artefatos reais coletados na Web. Tal ferramenta poderá servir tanto como prova de conceito da proposta quanto como base para investigações futuras sobre segurança de aplicações Web que utilizam *WebAssembly*.

Referências Bibliográficas

Ahmadi, S. (2025). Evolving botnet defenses: A survey of machine learning approaches for identifying polymorphic and evasive malware. *International Journal of Innovative Research and Scientific Studies*, 8(2):338–347.

Almurshid, H. A., Almomani, I., Khalifa, M. A., and El-Shafai, W. (2024). A holistic intelligent cryptojacking malware detection system. *IEEE Access*, 12:161417–161433.

Bai, S., Kolter, J. Z., and Koltun, V. (2018). An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*.

Berrios, S., Leiva, D., Olivares, B., and Hermosilla, P. (2025). Systematic review: Malware detection and classification in cybersecurity. *Applied Sciences*, 15(14):7747.

Bhansali, S., Aris, A., Acar, A., Oz, H., and Uluagac, A. S. (2022). A first look at code obfuscation for webassembly. In *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '22, page 140–145, New York, NY, USA. Association for Computing Machinery.

Bian, W., Meng, W., and Zhang, M. (2020). Minethrottle: Defending against wasm in-browser cryptojacking. In *Proceedings of The Web Conference 2020*, pages 3112–3118.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

Bommasani, R. (2021). On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*.

Bordawekar, R. and Shmueli, O. (2017). Using word embedding to enable semantic queries in relational databases. In *Proceedings of the 1st workshop on data management for end-to-end machine learning*, pages 1–4.

Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

Cabrera-Arteaga, J., Monperrus, M., Toady, T., and Baudry, B. (2023). Webassembly diversification for malware evasion. *Computers & Security*, 131:103296.

Caldiera, V. R. B.-G. and Rombach, H. D. (1994). Goal question metric paradigm. *Encyclopedia of Software Engineering*, 1(528-532):6.

Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357.

Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794.

Coinbase (2025). https://www.coinbase.com/pt-br/learn/crypto-basics/what-is-mining. acessado em 27-Fev-2025.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20(3):273–297.

Crypto Potato (2023). Ataques de cryptojacking dispararam 400 porcentos no 1 trimestre de 2023: Relatorio da sonicwall. https://cryptopotato.com/cryptojacking-attacks-skyrocketed-by-400-in-h1-2023-sonicwall-report/. acessado em 17-Dez-2024.

Cside (2025). Cryptojacking is dead: long live cryptojacking. acessado em 15-10-2025.

Cyber Threat Alliance (2018). The illicit cryptocurrency mining threat. https://cyberthreatalliance.org/wp-content/uploads/2018/09/CTA-Illicit-CryptoMining-Whitepaper.pdf. acessado em 07-03-2025.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*, pages 4171–4186.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al. (2021). An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations (ICLR)*.

Eskandari, S., Leoutsarakos, A., Mursch, T., and Clark, J. (2018). A first look at browser-based cryptojacking. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 58–66.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. (2020). Codebert: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020*, pages 1536–1547.

Feng, Z. and Luo, Q. (2020). Evaluating memory-hard proof-of-work algorithms on three processors. *Proceedings of the VLDB Endowment*, 13(6):898-911.

Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232.

Géron, A. (2022). *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. "O'Reilly Media, Inc.".

Golsch, L. (2019). Webassembly: Basics. *Technical University of Braunschweig.-2019*.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al. (2020). Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Géron, A. (2022). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media, 3rd edition.

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. (2017). Bringing the web up to speed with webassembly. *SIGPLAN Not.*, 52(6):185–200.

Harnes, H. and Morrison, D. (2024a). Cryptic bytes: Webassembly obfuscation for evading cryptojacking detection. *arXiv preprint arXiv:2403.15197*.

Harnes, H. and Morrison, D. (2024b). Sok: Analysis techniques for webassembly. *Future Internet*, 16(3):84.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning.* Springer, 2nd edition.

Heinrich, T., Will, N. C., Obelheiro, R. R., and Maziero, C. A. (2024). A categorical data approach for anomaly detection in webassembly applications. In *ICISSP*, pages 275–284.

Hochreiter, S. (1998). The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116.

Hoffman, K. (2019). Programming webassembly with rust: unified development for web, mobile, and embedded applications. *Unkwon.*

Hong, G., Yang, Z., Yang, S., Zhang, L., Nan, Y., Zhang, Z., Yang, M., Zhang, Y., Qian, Z., and Duan, H. (2018). How you get shot in the back: A systematical study about cryptojacking in the real world. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1701–1713, New York, NY, USA. Association for Computing Machinery.

Hosmer Jr, D. W., Lemeshow, S., and Sturdivant, R. X. (2013). *Applied logistic regression.* John Wiley & Sons.

Huang, H. and Zhao, J. (2024). Multi-modal learning for webassembly reverse engineering. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 453–465.

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2021). *An Introduction to Statistical Learning.* Springer, 2nd edition.

Jegou, H., Douze, M., and Schmid, C. (2010). Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128.

Jelodar, H., Bai, S., Hamedi, P., Mohammadian, H., Razavi-Far, R., and Ghorbani, A. (2025). Large language model (llm) for software security: Code analysis, malware analysis, reverse engineering. *arXiv preprint arXiv:2504.07137.*

Kaza, V. S., Alamer, A., Alam, S., Shuaib, M., Basudan, S., Sheneamer, A., et al. (2024). Detecting and forecasting cryptojacking attack trends in internet of things and wireless sensor networks devices. *PeerJ Computer Science*, 10:e2491.

Kharraz, A., Ma, Z., Murley, P., Lever, C., Mason, J., Miller, A., Borisov, N., Antonakakis, M., and Bailey, M. (2019). Outguard: Detecting in-browser covert cryptocurrency mining in the wild. In *The World Wide Web Conference*, WWW '19, page 840–852, New York, NY, USA. Association for Computing Machinery.

Kim, M., Jang, H., and Shin, Y. (2022). Avengers, assemble! survey of webassembly security solutions. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 543–553.

Kitchenham, B., Charters, S., et al. (2007). Guidelines for performing systematic literature reviews in software engineering.

Konoth, R. K., Vineti, E., Moonsamy, V., Lindorfer, M., Kruegel, C., Bos, H., and Vigna, G. (2018). Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1714–1730, New York, NY, USA. Association for Computing Machinery.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

Liaw, A. and Wiener, M. (2002). Classification and regression by randomforest. *R News*, 2(3):18–22.

Lonkar, A. and Chandrayan, S. (2018). The dark side of webassembly. *Accessed 27th-October-2021. URL: https://www. virusbulletin. com/virusbulletin/2018/10/dark-side-webassembly [cited 27th October 2021]*.

Loose, N., Mächtle, F., Pott, C., Bezsmertnyi, V., and Eisenbarth, T. (2023). Madvex: instrumentation-based adversarial attacks on machine learning malware detection. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 69–88. Springer.

Ma, W., Liu, S., Lin, Z., Wang, W., Hu, Q., Liu, Y., Zhang, C., Nie, L., Li, L., and Liu, Y. (2023). Lms: Understanding code syntax and semantics for code analysis. *arXiv preprint arXiv:2305.12138*.

Malkov, Y. A. and Yashunin, D. A. (2018). Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836.

Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., et al. (2020). Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 1(3):3.

Manthena, H., Shajarian, S., Kimmel, J. C., Abdelsalam, M., Khorsandroo, S., and Gupta, M. (2025a). Explainable artificial intelligence (XAI) for malware analysis: A survey of techniques, applications, and open challenges. *IEEE Access*.

Manthena, H., Shajarian, S., Kimmell, J. C., Abdelsalam, M., Khorsandroo, S., and Gupta, M. (2025b). Explainable artificial intelligence (xai) for malware analysis: A survey of techniques, applications, and open challenges. *IEEE Access*, 13:61611–61640.

Marais, B., Quertier, T., and Barrue, G. (2025). Semantic preprocessing for llm-based malware analysis. *arXiv preprint arXiv:2506.12113*.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013b). Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26.

Musch, M., Wressnegger, C., Johns, M., and Rieck, K. (2019). Thieves in the browser: Web-based cryptojacking in the wild. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ARES '19, New York, NY, USA. Association for Computing Machinery.

Naseem, F. N., Aris, A., Babun, L., Tekiner, E., and Uluagac, A. S. (2021). Minos: A lightweight real-time cryptojacking detection system. In *NDSS*.

Nielsen, D. (2016). Tree boosting with xgboost – why does xgboost win "every"machine learning competition? Master's thesis, Norwegian University of Science and Technology (NTNU).

Pan, J. J., Wang, J., and Li, G. (2023). Survey of vector database management systems. *arXiv preprint arXiv:2310.14021*.

Peralta, A. D., dos Santos, E. M., Gordo, M., Xie, J., and Colonna, J. G. (2025). Improving soundscape retrieval for bioacoustic monitoring: An analysis of fusion techniques with pre-trained embeddings. In *Brazilian Symposium on Multimedia and the Web (WebMedia)*, pages 312–320. SBC.

Perrone, G. and Romano, S. P. (2025). Webassembly and security: a review. *Computer Science Review*, 56:100728.

Petticrew, M. and Roberts, H. (2008). *Systematic reviews in the social sciences: A practical guide*. John Wiley & Sons.

Pouyanfar, S., Sadiq, S., Yan, Y., Tian, H., Tao, Y., Reyes, M. P., Shyu, M.-L., Chen, S.-C., and Iyengar, S. S. (2018). A survey on deep learning: Algorithms, techniques, and applications. *ACM computing surveys (CSUR)*, 51(5):1–36.

Qdrant Tech (2026). Qdrant. acessado em 26-03-2026.

Ray, P. P. (2023). An overview of webassembly for iot: Background, tools, state-of-the-art, challenges, and future directions. *Future Internet*, 15(8):275.

Romano, A., Zheng, Y., and Wang, W. (2020). Minerray: Semantics-aware analysis for ever-evolving cryptojacking detection. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1129–1140.

Rossberg, A., Haas, A., Schuff, D., Titzer, B., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. (2024). Webassembly core specification. `https://webassembly.github.io/spec/core/`. W3C Community Group Draft.

Saad, M. and Mohaisen, D. (2024). Analyzing in-browser cryptojacking. *IEEE Transactions on Dependable and Secure Computing*, 21(6):5448–5460.

Schölkopf, B. and Smola, A. J. (2002). *Learning with Kernels*. MIT Press.

She, X., Zhao, Y., and Wang, H. (2024). Wadec: Decompiling webassembly using large language model. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 481–492.

Sonic Wall (2024). Sonicwall cyber threat report. https://www.sonicwall.com/threat-report. acessado em 17-Dez-2024.

Taipalus, T. (2024). Vector database management systems: Fundamental concepts, use-cases, and current challenges. *Cognitive Systems Research*, 85:101216.

Taipalus, T., Grahn, H., Turtiainen, H., and Costin, A. (2024). Utilizing vector database management systems in cyber security. In *Proceedings of the 23th European Conference on Cyber Warfare and Security. Academic Conferences International Ltd*.

Taylor, M. B. (2017). The evolution of bitcoin hardware. *Computer*, 50(9):58–66.

Varlioglu, S., Gonen, B., Ozer, M., and Bastug, M. (2020). Is cryptojacking dead after coinhive shutdown? In *2020 3rd International Conference on Information and Computer Technologies (ICICT)*, pages 385–389.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., et al. (2017). Attention is all you need [j]. *Advances in neural information processing systems*, 30(1):261–272.

W3C (2024). https://webassembly.org/. acessado em 07-09-2024.

Wang, H., Zhang, Y., Pan, X., and Huang, W. (2023). Nocrypto: A web mining behavior detection method based on rgb images. In *International Conference on Security and Privacy in New Computing Environments*, pages 224–236. Springer.

Wang, W., Ferrell, B., Xu, X., Hamlen, K. W., and Hao, S. (2018). Seismic: Secure in-lined script monitors for interrupting cryptojacks. In Lopez, J., Zhou, J., and Soriano, M., editors, *Computer Security*, pages 122–142, Cham. Springer International Publishing.

Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., et al. (2022). Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*.

Wu, M.-H., Lai, Y.-J., Hwang, Y.-L., Chang, T.-C., and Hsu, F.-H. (2022). Minerguard: A solution to detect browser-based cryptocurrency mining through machine learning. *Applied Sciences*, 12(19):9838.

Xia, Y., He, P., Zhang, X., Liu, P., Ji, S., and Wang, W. (2023). Static semantics reconstruction for enhancing javascript-webassembly multilingual malware detection. In *European Symposium on Research in Computer Security*, pages 255–276. Springer.