



Mitigação de Ataques à Cadeia de Abastecimento JavaScript/npm através de Isolamento Granular baseado em WebAssembly

RAFAEL JOSÉ BRANDÃO GOMES

Junho de 2026

Mitigação de Ataques à Cadeia de Abastecimento JavaScript/npm através de Isolamento Granular baseado em WebAssembly

Rafael Gomes

**Dissertação para obtenção do Grau de Mestre em
Engenharia Informática, Área de Especialização em
Engenharia de Software**

Orientador: Dr. Nuno Pereira

Porto, 29 de junho de 2026

Declaração de Integridade

Declaro ter conduzido este trabalho académico com integridade.

Não plagiei ou apliquei qualquer forma de uso indevido de informações ou falsificação de resultados ao longo do processo que levou à sua elaboração.

Portanto, o trabalho apresentado neste documento é original e de minha autoria, não tendo sido utilizado anteriormente para nenhum outro fim. As exceções estão explicitamente reconhecidas na secção “Considerações éticas” do primeiro capítulo. Esta secção também declara como as ferramentas de IA foram utilizadas e para que finalidade.

Declaro ainda que tenho pleno conhecimento do Código de Conduta Ética do P.PORTO.

ISEP, Porto, 29 de junho de 2026

Resumo

O desenvolvimento de software moderno possui uma forte dependência de bibliotecas de terceiros, estimando-se que uma grande parte do código em aplicações atuais provém de fontes externas, com destaque para o ecossistema JavaScript/npm . Esta dependência, embora benéfica para a produtividade, introduz riscos de segurança, nomeadamente através de ataques à cadeia de abastecimento. O problema central reside no modelo de execução permissivo dos ambientes JavaScript, onde o código de terceiros herda implicitamente os privilégios da aplicação hospedeira, operando sem o isolamento necessário para conter dependências comprometidas.

Neste contexto, esta dissertação concebeu e concretizou um protótipo de isolamento granular para bibliotecas JavaScript executadas no browser. A solução integra-se no processo de construção através de um plugin Webpack, gera *proxies* para preservar a forma habitual de importação e executa cada biblioteca selecionada num contexto QuickJS alojado em WebAssembly. O *runtime* aplica uma política de negação por omissão, expondo apenas capacidades explicitamente autorizadas e mediando a comunicação entre a aplicação e a biblioteca isolada através de membranas e marshallers extensíveis.

A avaliação analisou compatibilidade, desempenho e segurança. Foram reutilizadas suites de teste de dez bibliotecas, das quais sete preservaram mais de 80% da respetiva execução de referência nativa. No browser, dez cargas de trabalho executadas em Chromium mostraram que o isolamento introduz custos relevantes, dependentes do volume e da frequência das travessias entre contextos. Em cenários com maior comunicação, os adaptadores de carga de trabalho reduziram o sobrecusto de execução da biblioteca Lodash de 1350,59× para 78,41× e da Zod de 623,19× para 16,21×. Em três cenários de segurança, envolvendo poluição de protótipos, interceção de Application Programming Interfaces (APIs) sensíveis e acesso a uma capacidade não autorizada, a *sandbox* preservou o comportamento legítimo e mitigou os efeitos observados na execução nativa. Os resultados demonstram a viabilidade técnica da abordagem, embora evidenciem que a sua aplicação prática depende do perfil da biblioteca, das permissões concedidas e do padrão de comunicação com a aplicação.

Palavras-chave: WebAssembly, JavaScript, Software Supply Chain Security, Granular Isolation, npm, Sandboxing

Abstract

Modern software development relies heavily on third-party libraries, with an estimated large portion of the code in current applications coming from external sources, particularly the JavaScript/npm ecosystem. This dependency, while beneficial to productivity, introduces security risks, notably through supply chain attacks. The central problem lies in the permissive execution model of JavaScript environments, where third-party code implicitly inherits the privileges of the host application, operating without the necessary isolation to contain compromised dependencies.

In this context, this dissertation designed and implemented a prototype for granular isolation of JavaScript libraries executed in the browser. The solution integrates with the build process through a Webpack plugin, generates proxies to preserve the usual import style, and executes each selected library in a QuickJS context hosted in WebAssembly. At runtime, it applies a default-deny policy, exposes only explicitly authorised capabilities, and mediates communication between the host application and the isolated library through membranes and extensible marshallers.

The evaluation analysed compatibility, performance, and security. Test suites from ten libraries were reused, with seven preserving more than 80% of their native baseline. In the browser, ten workloads executed in Chromium showed that isolation introduces relevant costs, depending on the volume and frequency of crossings between contexts. In communication-heavy scenarios, workload adapters reduced the overhead of Lodash from $1350.59\times$ to $78.41\times$ and of Zod from $623.19\times$ to $16.21\times$. In three security scenarios, covering prototype pollution, interception of sensitive APIs, and access to an unauthorised capability, the sandbox preserved legitimate behaviour and mitigated the effects observed in native execution. The results demonstrate the technical feasibility of the approach, while showing that its practical use depends on the library profile, the granted permissions, and the communication pattern with the application.

Agradecimentos

Gostaria de expressar a minha gratidão a todos os que me apoiaram ao longo desta dissertação. Este trabalho não seria possível sem a contribuição e o incentivo de muitas pessoas e instituições, às quais estou profundamente agradecido.

Primeiramente, desejo agradecer ao meu orientador, Doutor Nuno Pereira, cuja orientação, conhecimento e constante disponibilidade foram determinantes para a concretização deste trabalho. O seu espírito crítico e sugestões ao longo das diferentes fases da investigação contribuíram decisivamente para a qualidade do resultado alcançado.

Agradeço igualmente ao Instituto Superior de Engenharia do Porto e aos docentes do Mestrado em Engenharia de Software pelos conhecimentos transmitidos, pela exigência académica e pelo espírito crítico fomentado ao longo deste percurso, que contribuíram de forma decisiva para a realização deste trabalho.

Aos meus colegas de curso, agradeço pelas valiosas discussões, partilha de conhecimento e apoio ao longo de todo este percurso. Cada conversa, cada debate e cada colaboração contribuíram não só para a minha formação académica, mas também para o meu crescimento pessoal.

Dedico um agradecimento especial à minha família e amigos. Aos meus pais, o meu mais profundo agradecimento pelo incansável incentivo e pelo apoio incondicional ao longo de todo o meu percurso académico. A sua confiança e a sua presença foram a base que me sustentou nos momentos de dificuldade e a inspiração para continuar o meu trajeto.

Por fim, estendo o meu agradecimento a todos aqueles que, direta ou indiretamente, contribuíram para a concretização desta dissertação. A todos, o meu sincero obrigado.

Índice

Lista de Figuras	xv
Lista de Tabelas	xvii
Lista de Código	xix
Glossário	xxi
Lista de Acrônimos	xxiii
1 Introdução	1
1.1 Contexto	1
1.2 Problema	3
1.3 Questões de Investigação e Objetivos	4
1.3.1 Pergunta Principal de Investigação	5
Subperguntas de Investigação	5
1.3.2 Objetivos	5
Objetivo Geral	5
Objetivos Específicos	5
1.3.3 Principais Contributos Esperados	6
1.4 Metodologia	6
1.5 Considerações Éticas e Profissionais	7
1.6 Estrutura do Documento	7
2 Contexto	9
2.1 Modelo de Ameaças do Ecossistema JavaScript/npm	9
2.1.1 Execução com Privilégios Totais e Ausência de Isolamento	11
2.1.2 Herança Baseada em Protótipos e Mutação em Tempo de Execução	11
2.1.3 Sistema de Módulos Dinâmico e Acesso Irrestrito a Recursos	13
2.1.4 Mecanismos de Mitigação no Ecossistema npm e seus Limites	14
2.2 WebAssembly: Fundamentos e Características	15
2.2.1 Arquitetura e Modelo de Execução	15
2.2.2 Desempenho e Eficiência	17
2.2.3 Component Model	18
3 Estado da Arte	21
3.1 Método de Pesquisa	21
3.2 Integração entre JavaScript e WebAssembly	23
3.2.1 jco	24
3.2.2 QuickJS e Bibliotecas Relacionadas	25
3.3 Técnicas de Isolamento de Software	26

3.3.1	Isolamento Baseado em Processos e Contentores	27
3.3.2	Isolamento com WebAssembly	28
3.3.3	Isolamento para Ambientes de Execução JavaScript	28
	Isolamento no Lado do Servidor	28
	Isolamento no Lado do Cliente	29
	ShadowRealm	30
3.4	Comparação e Posicionamento da Solução	30
3.4.1	WebAssembly vs V8 Isolates	31
3.4.2	WebAssembly vs Cage4Deno	32
3.4.3	WebAssembly vs JSand	33
4	Análise	37
4.1	Requisitos Funcionais	37
4.2	Requisitos Não Funcionais	38
4.2.1	Functionality / Segurança	38
4.2.2	Usability / Usabilidade	39
4.2.3	Performance / Desempenho	40
4.2.4	Supportability / Manutenção e Evolução	40
4.2.5	Restrições	41
4.3	Casos de Uso	41
4.3.1	UC-01: Configurar Biblioteca para Isolamento	42
4.3.2	UC-02: Usar Biblioteca Isolada na Aplicação	42
4.3.3	UC-03: Conceder Capacidade Específica a uma Biblioteca Isolada	42
4.4	Síntese da Análise	43
5	Design da Solução	45
5.1	Visão Geral da Arquitetura	45
5.2	Arquitetura em Build-time	50
5.3	Arquitetura em Runtime	51
5.4	Modelo de Permissões e Membranas	54
5.5	Melhorias Futuras e Extensões da Conceção	58
5.6	Síntese das Decisões Arquiteturais	60
6	Implementação	63
6.1	Implementação do Plugin do Processo de Construção	63
6.2	Geração e Conteúdo dos <i>Proxies</i>	65
6.3	Implementação do Runtime da <i>sandbox</i>	66
6.4	Permissões, Membranas e Marshallers	69
6.4.1	Exposição de Capacidades Autorizadas	69
6.4.2	Membrana Host-VM	69
6.4.3	Membrana VM-Host	70
6.4.4	Marshallers Personalizados	71
6.5	Implementação do Modo Coarse-Grained	73
6.6	Síntese da Implementação	75
7	Avaliação	77
7.1	Avaliação de Compatibilidade	77
7.1.1	Método Experimental	77
7.1.2	Resultados de Compatibilidade	78
7.1.3	Discussão das Causas de Incompatibilidade	79

7.1.4	Síntese dos Resultados	80
7.2	Avaliação de Desempenho no Browser	80
7.2.1	Método Experimental	80
7.2.2	Resultados de Tempo de Execução	82
7.2.3	Influência das Membranas	83
7.2.4	Correlação com a Comunicação entre Contextos	86
7.2.5	Adaptadores de Carga de Trabalho	88
7.2.6	Síntese dos Resultados	90
7.3	Avaliação de Segurança	90
7.3.1	Método Experimental	90
7.3.2	Confinamento de Poluição de Prototype com Lodash	91
7.3.3	Interceção de APIs com strip-ansi	93
7.3.4	Permissões Granulares com ImageJS	94
7.3.5	Limitação na Transferência de Capacidades	95
7.3.6	Síntese dos Resultados	96
8	Conclusão	97
8.1	Síntese do Trabalho Desenvolvido	97
8.2	Resposta às Questões de Investigação	98
8.3	Cumprimento dos Objetivos e Requisitos	100
8.4	Principais Contributos	102
8.5	Limitações Identificadas	102
8.6	Trabalho Futuro	104
8.7	Considerações Finais	104
	Bibliografia	105
	Apêndice A Capacidades Web Candidatas a Exposição Controlada	111

Lista de Figuras

2.1	Módulo WebAssembly (Kim, Jang e Shin 2022)	16
2.2	Modelo de Memória Linear de um módulo WebAssembly (Kim, Jang e Shin 2022)	17
2.3	Ilustração conceptual do Modelo de Componentes WebAssembly. O diagrama demonstra como dois componentes isolados, desenvolvidos em linguagens distintas (Rust e Python), comunicam através de uma interface WIT.	19
3.1	Diagrama PRISMA para a pesquisa sobre o ecossistema JavaScript/npm	22
3.2	Diagrama PRISMA para a pesquisa sobre WebAssembly e isolamento	23
3.3	Arquitetura do Cage4Deno (Abbadini et al. 2023)	33
3.4	Arquitetura do JSand (Agten et al. 2012)	35
5.1	Vista lógica de componentes da solução	46
5.2	Vista de implementação da solução	48
5.3	Vista de implantação da solução	49
5.4	Sequência do processo de construção da solução	51
5.5	Sequência de inicialização e utilização da arquitetura em runtime	53
5.6	Acesso autorizado a uma capacidade do ambiente hospedeiro	56
5.7	Mediação de uma chamada da aplicação para a biblioteca isolada	57
5.8	Sequência de execução no modo <i>coarse-grained</i>	59
7.1	Razão entre o tempo mediano do cenário em <i>sandbox</i> e em execução nativa. As barras de erro representam o intervalo de confiança de 95% obtido por bootstrap.	82
7.2	Distribuições <i>bootstrap</i> do sobrecusto de Marked, TweetNaCl, strip-ansi, date-fns, ImageJS e MIME, apresentadas numa escala comum até 100×.	84
7.3	Distribuições <i>bootstrap</i> do sobrecusto de Luxon, Lodash, Zod e Yup, apresentadas numa escala própria por serem os quatro maiores <i>outliers</i> de sobrecusto de execução.	85
7.4	Sobrecusto temporal em função dos bytes estimados transferidos pelas membranas.	87
7.5	Diferença do pico de <i>heap</i> JavaScript em função dos bytes estimados transferidos pelas membranas.	87
7.6	Comparação do sobrecusto do modo transparente e dos adaptadores de carga de trabalho de Lodash e Zod.	89
7.7	Resultados do cenário de poluição de protótipos com Lodash.	92
7.8	Resultados da réplica controlada do ataque à cadeia de abastecimento com strip-ansi.	93
7.9	Resultados do cenário de permissões granulares com ImageJS.	95

Lista de Tabelas

1.1	Indicadores da escala do ecossistema JavaScript/npm	3
2.1	Âmbito das ameaças consideradas no trabalho	10
2.2	Vulnerabilidades estruturais do ecossistema JavaScript/npm	14
3.1	Comparação entre <i>jco</i> e QuickJS para integração JavaScript-WebAssembly	26
3.2	Comparação entre técnicas de isolamento	36
4.1	Requisitos funcionais da solução	38
4.2	Requisitos não funcionais de segurança	39
4.3	Requisitos não funcionais de usabilidade	39
4.4	Requisitos não funcionais de desempenho	40
4.5	Requisitos não funcionais de manutenção e evolução	40
4.6	Restrições da solução	41
4.7	Casos de uso principais	41
5.1	Capacidades ilustradas no exemplo de configuração	55
5.2	Síntese das decisões arquiteturais de conceção	61
6.1	Alinhamento entre conceção e implementação	76
7.1	Testes executados, testes passados e rácio de compatibilidade por biblioteca.	78
7.2	Tempo mediano por execução do cenário e sobrecusto da <i>sandbox</i>	83
7.3	Diagnóstico das membranas nas cargas de trabalho com maior sobrecusto.	84
7.4	Correlações de Spearman entre comunicação nas membranas e sobrecusto.	86
7.5	Síntese dos cenários de avaliação de segurança.	91
8.1	Síntese do cumprimento dos requisitos da solução	101
A.1	Capacidades Web candidatas a exposição controlada	112

Lista de Código

5.1	Exemplo declarativo de permissões por biblioteca	54
6.1	Exemplo de configuração de uma biblioteca no modo transparente	63
6.2	Redirecionamento de importações para o <i>proxy</i> gerado	64
6.3	Criação do bundle isolado da biblioteca	64
6.4	Estrutura inicial de um <i>proxy</i> gerado	65
6.5	Exportação nomeada gerada com suporte a chamada e construção	66
6.6	Sequência principal de inicialização da <i>sandbox</i>	68
6.7	Funções auxiliares pré-compiladas usadas pela Membrana VM-Host	68
6.8	Ligação de métodos nativos antes da exposição ao ambiente isolado	70
6.9	Chamada de método sobre objeto mantido no contexto QuickJS	71
6.10	Composição dos marshallers na configuração da Arena	72
6.11	Exemplo de configuração com adaptador de carga de trabalho	73
6.12	Ligação da carga de trabalho à biblioteca já carregada na <i>sandbox</i>	74
6.13	Forma simplificada de um <i>proxy</i> de carga de trabalho	74
6.14	Exemplo reduzido de adaptador de carga de trabalho para lodash	75
7.1	Exemplo reduzido de carga de trabalho transparente da biblioteca Lodash	81
7.2	Exemplo reduzido de adaptador de carga de trabalho da biblioteca Lodash	88
7.3	<i>Canary</i> usada no cenário Lodash	92
7.4	Interceção inofensiva de fetch no patch de strip-ansi	93
7.5	Tentativa de acesso adicional introduzida em ImageJS	94

Glossário

<i>bootstrap</i>	Técnica estatística de reamostragem com reposição utilizada para estimar a distribuição de uma estatística e calcular medidas de incerteza, como intervalos de confiança.
<i>canary</i>	Valor ou propriedade sentinela introduzida num cenário de teste para detetar se uma operação modificou estado que deveria permanecer inalterado.
<i>child compiler</i> (compilador filho)	Compilador auxiliar criado pelo Webpack dentro de uma compilação principal para produzir um artefacto separado, reutilizando a resolução e a configuração do processo de construção.
<i>closure</i>	Função que preserva acesso ao ambiente léxico onde foi criada, mesmo quando é invocada posteriormente noutro ponto do programa.
<i>hook</i>	Ponto de extensão disponibilizado por uma ferramenta ou plataforma para executar lógica própria numa fase específica do seu ciclo de vida.
<i>hot path</i>	Caminho de execução percorrido com frequência suficiente para influenciar de forma relevante o desempenho observado.
<i>internal slot</i>	Estado interno especificado pelo ECMAScript para certos objetos nativos, inacessível diretamente por código JavaScript mas validado por métodos desses objetos.
<i>outlier</i>	Observação atípica que difere significativamente da restante distribuição dos dados.
Proxy	API JavaScript que permite interpor operações sobre um objeto, como acesso a propriedades, chamada de função ou construção.
<i>realm</i>	Ambiente ECMAScript com objeto global, intrínsecos e cadeias de protótipos próprios.

sandbox

Ambiente de execução isolado que restringe a autoridade do código executado e medeia o acesso a capacidades externas.

wrapper

Objeto ou função intermediária que envolve outro valor para adaptar, restringir ou mediar o seu comportamento observável.

Lista de Acrónimos

ABI	Application Binary Interface.
API	Application Programming Interface.
CDP	Chrome DevTools Protocol.
cgroups	Control Groups.
CPU	Central Processing Unit.
DOM	Document Object Model.
DSR	Design Science Research.
I/O	Input/Output.
IA	Inteligência Artificial.
JIT	Just-In-Time.
MDN	Mozilla Developer Network.
OIDC	OpenID Connect.
SCA	Software Composition Analysis.
SLSA	Supply-chain Levels for Software Artifacts.
VM	Máquina Virtual.
Wasm	WebAssembly.
WIT	WebAssembly Interface Types.

Capítulo 1

Introdução

O presente capítulo estabelece os fundamentos deste trabalho, situando-o no panorama atual da Engenharia de Software e definindo o âmbito da investigação. O seu propósito é guiar o leitor desde a compreensão do cenário tecnológico até à definição dos objetivos que guiam o trabalho.

O capítulo inicia-se na Secção 1.1, onde se enquadra a relevância crescente do ecossistema JavaScript e dos registos de pacotes como o npm na infraestrutura digital moderna. Esta secção exemplifica ainda um ataque recente à cadeia de abastecimento JavaScript/npm, descrevendo o seu mecanismo de compromisso e o impacto potencial em aplicações Web. Esta contextualização é essencial para compreender a magnitude do impacto que as vulnerabilidades nesta cadeia de abastecimento podem provocar do lado do cliente.

Consequentemente, a Secção 1.2 afunila a discussão para identificar as lacunas de segurança específicas que motivam este trabalho. Aqui, a ausência de mecanismos de isolamento eficazes na execução de código de terceiros é apresentada não apenas como uma falha técnica, mas como um risco sistémico que carece de mitigação.

Caracterizado o problema, a Secção 1.3 formaliza o propósito do estudo. Este segmento apresenta a pergunta central que se procura responder, bem como os objetivos gerais e específicos que estruturam a proposta de solução baseada em WebAssembly. Adicionalmente, identificam-se os principais contributos esperados.

Para garantir a validade científica do processo, a Secção 1.4 descreve a abordagem de investigação adotada. Justifica-se aqui a escolha dos métodos que suportam tanto a revisão da literatura como a conceção e validação do artefacto proposto, assegurando o rigor necessário.

Na Secção 1.5 são apresentadas as considerações éticas e profissionais que guiam o desenvolvimento do trabalho. São indicados os códigos de conduta seguidos, considerações acerca dos dados utilizados e segurança do desenvolvimento, e a descrição das competências a desenvolver.

Por fim, a Secção 1.6 fornece uma síntese da estrutura do restante trabalho.

1.1 Contexto

Em 8 de setembro de 2025, foram publicadas versões maliciosas de 18 pacotes npm amplamente utilizados, entre os quais `debug`, `chalk`, `ansi-styles`, `supports-color`, `strip-ansi`,

`ansi-regex`, `wrap-ansi` e `color-convert`¹ (Eriksen 2025; Slavin e Kuznicki 2025). Em conjunto, estes pacotes registavam mais de dois mil milhões de downloads semanais (Eriksen 2025; Slavin e Kuznicki 2025). O ataque teve origem no comprometimento da conta de um dos seus principais responsáveis pela manutenção, após uma campanha de *phishing* que utilizou o falso domínio de suporte `npmjs.help`. Com acesso à conta, os atacantes conseguiram publicar no registo npm novas versões dos pacotes contendo o código malicioso (Eriksen 2025; Slavin e Kuznicki 2025).

O código injetado foi concebido especificamente para execução no browser. A carga maliciosa detetava a presença de objetos Web, interceptava comunicações efetuadas através de `fetch` e `XMLHttpRequest` e manipulava Application Programming Interfaces (APIs) de carteiras de criptomoedas, incluindo `window.ethereum`. Ao inspecionar pedidos, respostas e operações Web3, podia substituir destinatários ou autorizações e redirecionar fundos para endereços controlados pelos atacantes, mantendo a interface apresentada ao utilizador aparentemente inalterada (Eriksen 2025; Sharma 2025; Slavin e Kuznicki 2025).

Este incidente ilustra o problema central deste trabalho: uma dependência npm, quando integrada numa aplicação Web e executada no browser, pode herdar capacidades do ambiente hospedeiro e realizar operações sensíveis sem uma fronteira arquitetural explícita. O ataque constitui, assim, uma manifestação concreta do impacto que um compromisso da cadeia de abastecimento pode produzir diretamente do lado do cliente, não sendo um caso isolado.

Isto é agravado pelo facto do desenvolvimento de software moderno assentar numa grande dependência de código desenvolvido por terceiros. Estimativas indicam que até 90% do código integrado numa aplicação atual é importado através de bibliotecas externas (Sonatype 2024), particularmente no ecossistema JavaScript/npm, o maior registo de pacotes do mundo com aproximadamente 4.8 milhões de projetos e 4.5 triliões de pedidos anuais (Sonatype 2024).

Ataques à cadeia de abastecimento exploram esta dependência para inserir código malicioso em bibliotecas amplamente utilizadas. O número de pacotes maliciosos identificados nos principais ecossistemas atingiu 512.847 num ano, representando um aumento de 156% (Sonatype 2024).

A Tabela 1.1 sintetiza os principais indicadores usados para enquadrar a escala do problema no ecossistema JavaScript/npm.

¹A lista inicial identificada compreendia `backslash`, `chalk-template`, `supports-hyperlinks`, `has-ansi`, `simple-swizzle`, `color-string`, `error-ex`, `color-name`, `is-arrayish`, `slice-ansi`, `color-convert`, `wrap-ansi`, `ansi-regex`, `supports-color`, `strip-ansi`, `chalk`, `debug` e `ansi-styles`.

Tabela 1.1: Indicadores da escala do ecossistema JavaScript/npm

Dimensão	Dado
Código externo	Até 90% do código de uma aplicação moderna é composto por bibliotecas de terceiros.
Downloads anuais	O npm serve 4.5 triliões de pedidos anuais e aproximadamente 4.8 milhões de projetos.
Alcance potencial no browser	O ataque de setembro de 2025 comprometeu 18 pacotes npm com mais de dois mil milhões de downloads semanais agregados.
Aumento de ataques	Foram identificados 512.847 pacotes maliciosos num único ano, representando um aumento de 156%.

O problema crítico manifesta-se na importação destas bibliotecas comprometidas pelas aplicações Web. O modelo de execução JavaScript em ambientes de browser opera num paradigma de confiança total, onde qualquer biblioteca importada executa com os mesmos privilégios do contexto hospedeiro (Zimmermann et al. 2019). Isto significa que código malicioso pode aceder livremente a APIs sensíveis do browser sem qualquer restrição arquitetural.

Sem isolamento granular, este código tem a capacidade de exfiltrar dados sensíveis, manipular páginas, ou comunicar com servidores externos para extrair credenciais. As defesas convencionais são predominantemente reativas e operam ao nível da deteção de vulnerabilidades conhecidas, falhando na prevenção de ataques que exploram permissões legítimas concedidas ao código de terceiros (Zahan et al. 2022; Zimmermann et al. 2019).

WebAssembly (Wasm) apresenta-se como uma tecnologia candidata para mitigar estas consequências através de isolamento granular em tempo de execução. Concebido para execução segura de código compilado na Web, Wasm oferece garantias arquiteturais ausentes no JavaScript: memória linear isolada por módulo, validação de tipos em tempo de compilação e isolamento nativo (Haas et al. 2017). Estudos empíricos demonstram desempenho próximo ao nativo e sobrecusto (*overhead*) inferior a 10% em cenários de *micro-sandboxing* (Narayan et al. 2020).

Este trabalho investiga em que medida uma arquitetura baseada em Wasm pode limitar o impacto de código malicioso resultante de ataques à cadeia de abastecimento JavaScript/npm, através de isolamento granular de bibliotecas de front-end, mantendo níveis aceitáveis de desempenho e usabilidade.

1.2 Problema

A dependência massiva de bibliotecas de terceiros no ecossistema JavaScript, gerida maioritariamente através do npm, introduziu uma superfície de ataque crítica nas aplicações modernas. O problema central reside no modelo de segurança permissivo dos ambientes de execução JavaScript tradicionais. Por desenho, estes ambientes operam sob um princípio de confiança implícita, onde qualquer *script* carregado na página — seja o código da própria aplicação ou uma dependência de terceiros — partilha o mesmo contexto global e herda os mesmos privilégios do utilizador (Zimmermann et al. 2019). Consequentemente, uma dependência maliciosa ou comprometida obtém acesso sem restrições, podendo exfiltrar dados

sensíveis, manipular páginas, ou comunicar com servidores externos, tal como evidenciado pelo ataque descrito na Secção 1.1.

Esta vulnerabilidade estrutural é exacerbada pela natureza transitiva das dependências. Uma aplicação típica importa centenas de pacotes indiretos, tornando a auditoria manual impraticável. Os ataques à cadeia de abastecimento de software exploram esta complexidade, injetando código malicioso em pacotes legítimos que são subsequentemente integrados em milhares de projetos a jusante (Sonatype 2024). As abordagens convencionais de mitigação, como a análise estática de código e a verificação de vulnerabilidades conhecidas (Software Composition Analysis (SCA)), revelam-se insuficientes contra vetores de ataque sofisticados. Técnicas de ofuscação de código e carregamento dinâmico de cargas maliciosas em tempo de execução permitem frequentemente contornar estas deteções estáticas (Romano et al. 2022).

No contexto do desenvolvimento de aplicações do lado do servidor, a indústria mitigou parcialmente este risco recorrendo a soluções de isolamento baseadas no sistema operativo, como contentores Linux ou máquinas virtuais. Embora eficazes na proteção do hospedeiro, estas tecnologias impõem um sobrecusto de recursos significativo e fornecem um isolamento de menor granularidade, concebido para segregar aplicações inteiras e não módulos individuais (Wiegratz 2024). No entanto, esta abordagem não tem paralelo viável para aplicações executadas no browser. O uso de mecanismos pesados de virtualização ou mesmo de *iFrames* para isolar cada dependência modular numa aplicação Web degradaria a experiência do utilizador devido à latência e ao consumo de memória (Narayan et al. 2020).

Existe, portanto, uma lacuna técnica na capacidade de impor limites de segurança granulares dentro do próprio processo da aplicação. A ausência de um mecanismo leve e eficiente, capaz de confinar dependências não confiáveis num ambiente isolado sem degradar severamente o desempenho de execução, impede a adoção de uma arquitetura de segurança de "confiança zero" no próprio browser. É neste contexto que a investigação sobre o uso de Wasm como mecanismo de isolamento em tempo de execução se torna premente, visando conciliar a segurança robusta com a eficiência necessária para ambientes de produção (Besozzi et al. 2025).

1.3 Questões de Investigação e Objetivos

A investigação parte do problema identificado na Secção 1.2 — a ausência de isolamento granular no ecossistema npm permite que código malicioso explore privilégios excessivos em tempo de execução. A formulação das questões de investigação segue uma progressão necessária: primeiro, é preciso caracterizar que vetores de ataque e que vulnerabilidades arquiteturais devem ser isolados; depois, compreender por que razão as técnicas convencionais não impedem suficientemente abusos em tempo de execução; em seguida, avaliar se o modelo de segurança do Wasm oferece propriedades adequadas para isolar código não confiável no browser e controlar o acesso a capacidades do ambiente hospedeiro, necessidade demonstrada pelo ataque mencionado na Secção 1.1; por fim, comparar essa abordagem com alternativas existentes quanto à eficácia do isolamento, desempenho e experiência de desenvolvimento. Para estruturar esta investigação, formula-se uma pergunta principal de investigação que é dividida em quatro subperguntas mais específicas, cada uma abordando uma dimensão específica do problema.

1.3.1 Pergunta Principal de Investigação

Em que medida uma arquitetura baseada em **WebAssembly** com isolamento granular pode mitigar eficazmente, no browser, o impacto dos vetores de ataque predominantes na cadeia de abastecimento JavaScript/npm, mantendo níveis aceitáveis de desempenho e usabilidade?

Subperguntas de Investigação

1. **RQ1:** Quais são os vetores de ataque predominantes e as vulnerabilidades arquiteturais críticas na cadeia de abastecimento de software JavaScript/npm?
2. **RQ2:** Quais as limitações das técnicas convencionais de isolamento na prevenção de ataques à cadeia de abastecimento em tempo de execução?
3. **RQ3:** Em que medida o modelo de segurança do WebAssembly oferece mecanismos eficazes para o isolamento granular de código não confiável?
4. **RQ4:** Como se compara o isolamento baseado em WebAssembly com outras técnicas de isolamento em termos de eficácia na mitigação, sobrecusto de desempenho e impacto na experiência de desenvolvimento?

1.3.2 Objetivos

Para responder às questões de investigação formuladas, o trabalho estrutura-se em torno de um objetivo geral — construir e avaliar uma arquitetura baseada em Wasm — que se operacionaliza através de seis objetivos específicos correspondentes às fases metodológicas: caracterização do problema (Obj. 1), análise das capacidades do Wasm para isolamento granular no browser (Obj. 2), definição dos requisitos (Obj. 3), conceção da solução (Obj. 4), implementação (Obj. 5) e validação de segurança, desempenho e usabilidade (Obj. 6)

Objetivo Geral

Investigar, conceber e avaliar uma abordagem baseada em WebAssembly para isolamento granular de dependências JavaScript/npm executadas no browser, com o propósito de reduzir o impacto de código de terceiros potencialmente malicioso em aplicações Web, preservando níveis aceitáveis de compatibilidade, desempenho e usabilidade para o desenvolvedor.

Objetivos Específicos

1. **Caracterizar** os riscos de segurança associados à cadeia de abastecimento JavaScript/npm com impacto em aplicações Web do lado do cliente, identificando vetores de ataque relevantes e limitações das práticas convencionais de mitigação.
2. **Analisar** em que medida o WebAssembly pode reduzir os privilégios de dependências não confiáveis no contexto de aplicações Web.
3. **Definir** os requisitos de uma solução de isolamento granular para bibliotecas de terceiros, considerando segurança, compatibilidade com bibliotecas reais, desempenho e esforço de adoção.
4. **Conceber** uma solução baseada em WebAssembly que permita executar dependências selecionadas em isolamento, controlando explicitamente o acesso a capacidades sensíveis do ambiente hospedeiro.

5. **Implementar** um protótipo funcional que demonstre a viabilidade técnica da abordagem proposta em cenários representativos de utilização de bibliotecas npm.
6. **Avaliar** a solução proposta quanto à sua eficácia de isolamento, compatibilidade funcional, impacto no desempenho e implicações práticas para a experiência de desenvolvimento.

1.3.3 Principais Contributos Esperados

O cumprimento dos objetivos traçados produzirá contributos seguintes:

- Catálogo de vetores de ataque à cadeia de abastecimento npm e mapeamento para mecanismos de defesa Wasm
- Arquitetura de referência para isolamento granular de dependências JavaScript com Wasm
- Prova de conceito demonstrando viabilidade técnica da interoperabilidade JS-Wasm em contexto de segurança
- Avaliação comparativa entre isolamento Wasm e técnicas convencionais
- Identificação de compromissos entre segurança, desempenho e usabilidade

1.4 Metodologia

A natureza deste projeto, focado na conceção e validação de um artefacto tecnológico inovador para resolver um problema prático identificado, enquadra-se na metodologia de Design Science Research (DSR). Esta abordagem é particularmente adequada para a Engenharia de Software, pois orienta a criação de artefactos que procuram satisfazer necessidades e resolver problemas relevantes num ambiente real.

O processo de investigação foi estruturado em ciclos iterativos que asseguram o rigor científico e a relevância prática, compreendendo as seguintes fases fundamentais:

1. **Identificação do Problema e Motivação:** Esta fase inicial, materializada nos capítulos introdutórios, focou-se na definição das vulnerabilidades na cadeia de abastecimento *npm* e na justificação da necessidade de uma abordagem de isolamento mais granular sem um impacto proibitivo no desempenho e usabilidade. A relevância do problema foi fundamentada através da análise de relatórios recentes de cibersegurança e da identificação das limitações nas soluções de isolamento atuais.
2. **Definição dos Objetivos da Solução:** Com base no problema identificado, estabeleceram-se os objetivos que o artefacto deve cumprir. O objetivo central é inferir uma arquitetura que garanta a segurança através do confinamento de dependências, sem introduzir uma penalização de desempenho e usabilidade proibitiva para ambientes de produção.
3. **Conceção e Desenvolvimento:** Nesta etapa, será concebida a arquitetura do sistema e implementado um protótipo funcional. Seguindo uma abordagem incremental descrita no planeamento do projeto, o desenvolvimento do protótipo será dividido em duas iterações principais:
 - **Iteração 1:** Implementação dos mecanismos nucleares de isolamento e da ponte de comunicação entre *JavaScript* e Wasm.

- **Iteração 2:** Otimização do desempenho e refinamento da usabilidade da solução.
4. **Demonstração e Avaliação:** A utilidade e eficácia do artefacto serão validadas experimentalmente. A avaliação será conduzida através de dois eixos principais:
 - **Eficácia de Segurança:** O protótipo será submetido a um conjunto de casos de teste que simulam vetores de ataque reais para verificar se o isolamento impede o compromisso do sistema anfitrião.
 - **Análise de Desempenho:** Serão realizados *benchmarks* para medir o sobrecusto introduzido pela camada de isolamento em métricas como tempo de execução e consumo de memória.
 5. **Comunicação:** Por fim, o processo culmina na documentação rigorosa de todo o ciclo de investigação, dos resultados obtidos e das contribuições para a área da Engenharia de Software, materializada na dissertação e na sua defesa pública.

1.5 Considerações Éticas e Profissionais

Este trabalho segue os princípios éticos estabelecidos no Regulamento do Código de Boas Práticas e de Conduta do Instituto Politécnico do Porto (Porto 2020), nomeadamente os artigos 6.^o (Deveres dos/as Estudantes), 8.^o (Declaração de Compromisso) e 10.^o (Boas práticas nas atividades de investigação). Adicionalmente, orienta-se pelos princípios do ACM Code of Ethics and Professional Conduct (Computing Machinery 2018) relativos à transparência, rigor científico e responsabilidade na divulgação de vulnerabilidades de segurança.

A investigação assegura integridade através de citação rigorosa de todas as fontes, registo transparente de metodologia e dados experimentais. Os resultados — positivos ou negativos — serão reportados sem omissão seletiva. Ferramentas de inteligência artificial foram utilizadas como apoio ao desenvolvimento do projeto, nomeadamente na exploração e comparação de alternativas tecnológicas, na discussão de opções de arquitetura e implementação, no auxílio à escrita e revisão de código, bem como na melhoria da redação do texto da dissertação. Todas as decisões técnicas e científicas, a conceção da solução, a implementação, a validação experimental e a redação final dos conteúdos são da responsabilidade do autor, tendo toda a informação produzida com o apoio destas ferramentas sido sujeita a análise e validação crítica.

O projeto não envolve participação de seres humanos, dados pessoais ou informação confidencial. A investigação utiliza exclusivamente bibliotecas npm públicas e dados de acesso aberto, respeitando sempre as licenças de utilização. Pacotes maliciosos foram manuseados de forma isolada, prevenindo riscos de propagação.

As competências técnicas necessárias foram desenvolvidas através da consulta de literatura académica e documentação oficial. As competências não técnicas cujo desenvolvimento foi identificado como mais relevante para o sucesso do projeto foram também devidamente identificadas e foi concebido um plano de ação específico para cada uma delas.

1.6 Estrutura do Documento

Este documento encontra-se organizado em oito capítulos principais, que acompanham o percurso da investigação desde a contextualização do problema até à conceção, implementação, avaliação e conclusão do trabalho. O Capítulo 2 apresenta o contexto do trabalho,

caracterizando o modelo de ameaças do ecossistema JavaScript/npm e introduzindo os fundamentos do Wasm relevantes para o isolamento de dependências. O Capítulo 3 apresenta o estado da arte, descrevendo o método de pesquisa adotado, analisando a integração entre JavaScript e Wasm e comparando técnicas de isolamento existentes, posicionando a abordagem baseada em Wasm face às alternativas atuais. O Capítulo 4 define a análise da solução, identificando os requisitos funcionais, os requisitos não funcionais e os casos de uso que orientam o desenvolvimento do artefacto. O Capítulo 5 descreve a conceção da solução, apresentando a arquitetura proposta, as principais decisões e a forma como estas respondem aos requisitos definidos. O Capítulo 6 detalha a implementação do protótipo, incluindo os mecanismos de integração com o processo de construção (*build process*), geração de *proxies*, execução isolada em QuickJS/Wasm e exposição controlada de capacidades do ambiente hospedeiro. O Capítulo 7 apresenta a avaliação da solução, estruturada em torno da compatibilidade funcional, impacto no desempenho e eficácia de isolamento perante cenários representativos. Por fim, o Capítulo 8 sintetiza os resultados obtidos, responde às questões de investigação, identifica limitações do trabalho e apresenta direções de evolução futura.

Capítulo 2

Contexto

O presente capítulo estabelece o enquadramento técnico e conceptual necessário para compreender o problema investigado. O objetivo é caracterizar os dois elementos que sustentam o trabalho: o modelo de ameaças associado ao ecossistema JavaScript/npm quando dependências de terceiros executam em aplicações Web no browser, e as propriedades da tecnologia WebAssembly.

Inicialmente, a Secção 2.1 analisa as vulnerabilidades estruturais do ecossistema JavaScript/npm relevantes para código incorporado no lado do cliente. Esta análise foca-se na execução com privilégios totais no contexto da página, na mutabilidade do modelo de protótipos, na opacidade introduzida por módulos dinâmicos e dependências transitivas, e nos limites dos mecanismos de mitigação disponibilizados pelo próprio ecossistema npm.

De seguida, a Secção 2.2 introduz os fundamentos do WebAssembly. São apresentados o seu modelo de execução, a memória linear isolada, as propriedades de desempenho e o Modelo de Componentes, estabelecendo a base para compreender por que motivo o WebAssembly pode ser usado como mecanismo de contenção em tempo de execução.

2.1 Modelo de Ameaças do Ecossistema JavaScript/npm

Esta secção apresenta o modelo de ameaças que motiva esta dissertação, com foco no impacto de dependências npm executadas no browser em aplicações Web do lado do cliente. O problema central não reside apenas na existência de pacotes vulneráveis ou maliciosos, mas no modo como o ecossistema JavaScript/npm executa dependências de terceiros: uma biblioteca importada pela aplicação tende a herdar a mesma autoridade do código hospedeiro, sem uma fronteira nativa que limite os recursos, o estado global ou as APIs que pode aceder (Staicu, Pradel e Livshits 2018; Zimmermann et al. 2019). Assim, uma dependência comprometida pode deixar de ser uma falha localizada e passar a constituir um ponto de compromisso da aplicação inteira.

A Tabela 2.1 delimita os vetores considerados no âmbito deste trabalho. Os detalhes técnicos de cada vetor são desenvolvidos nas subsecções seguintes.

Tabela 2.1: Âmbito das ameaças consideradas no trabalho

Ameaça	Âmbito	Mecanismo	Justificação
Código malicioso numa dependência npm no browser	Dentro	A dependência herda a autoridade da aplicação Web durante a execução.	Corresponde ao cenário central mitigado pelo isolamento em tempo de execução.
Abuso de APIs globais do browser	Dentro	O código acede ou interceta APIs como <code>fetch</code> , <code>XMLHttpRequest</code> ou <code>window.ethereum</code> .	Representa o impacto direto de uma dependência comprometida no lado do cliente.
Poluição de protótipos	Dentro	Propriedades injetadas em protótipos partilhados alteram o comportamento de objetos da aplicação.	Está dentro quando afeta dependências executadas no mesmo contexto JavaScript da aplicação.
Dependências transitivas e módulos dinâmicos	Fora do âmbito direto	A resolução dinâmica e a introdução indireta de pacotes ocultam parte da superfície de ataque.	São motivação estrutural, mas o trabalho não propõe deteção ou auditoria do grafo de dependências.
Scripts de instalação npm	Fora	Código executa durante a instalação, antes da execução da aplicação no browser.	O projeto atua sobre execução em tempo de execução no lado do cliente, não sobre a fase de instalação.
Código do lado do servidor	Fora	A dependência executa em ambientes como Node.js ou Deno, com capacidades do servidor.	O foco da dissertação é o confinamento de bibliotecas em aplicações Web no browser.

A análise organiza este problema em quatro dimensões complementares. A subsecção 2.1.1 avalia o princípio de autoridade ambiental e a consequente ausência de isolamento granular, características que permitem a dependências maliciosas herdar privilégios excessivos do ambiente hospedeiro. Seguidamente, a subsecção 2.1.2 aborda os riscos associados à herança baseada em protótipos, detalhando como a mutabilidade de objetos em tempo de execução viabiliza vetores de ataque complexos, como a poluição de protótipos. A subsecção 2.1.3 analisa a opacidade introduzida pelo sistema de módulos dinâmico e pela gestão de dependências transitivas, fatores que dificultam a análise estática e ocultam a superfície real de ataque das aplicações modernas. Por fim, a subsecção 2.1.4 contextualiza mecanismos de mitigação disponibilizados pelo ecossistema npm, clarificando o seu papel complementar face a mecanismos de isolamento em tempo de execução.

2.1.1 Execução com Privilégios Totais e Ausência de Isolamento

O aspecto fundamental que predispõe o ecossistema npm para ataques da cadeia de abastecimento é a ausência de isolamento granular no modelo de execução JavaScript. Quando uma aplicação Web importa uma biblioteca, o código dessa biblioteca executa com os mesmos privilégios que o código da aplicação, com acesso irrestrito aos mesmos recursos (Staicu, Pradel e Livshits 2018; Zimmermann et al. 2019). Esta característica significa que uma dependência não confiável consegue: aceder a APIs sensíveis do ambiente, modificar comportamentos críticos através de alterações ao estado global, e executar operações arbitrárias sem qualquer mecanismo nativo que limite o seu alcance.

A ausência de um modelo de capacidades baseado em princípios de "menor privilégio" significa que não existe uma forma nativa de conceder permissões granulares a cada dependência. Uma biblioteca executada no browser herda a autoridade disponibilizada à aplicação no contexto da página, sem que o seu acesso possa ser limitado a conjuntos específicos de recursos ou operações (Zimmermann et al. 2019). O ataque aos 18 pacotes npm descrito na Secção 1.1 demonstra a relevância desta característica arquitetural: após o comprometimento da conta de um responsável pela manutenção e a publicação das versões maliciosas, o código incorporado em aplicações Web conseguiu intercetar `fetch` e `XMLHttpRequest`, aceder a APIs de carteiras, como `window.ethereum`, e manipular transações no browser (Eriksen 2025; Sharma 2025; Slavin e Kuznicki 2025). Embora o comprometimento tenha ocorrido na cadeia de publicação, o seu impacto do lado do cliente resultou da autoridade que as aplicações concediam às dependências durante a execução.

A consequência prática desta configuração é que o impacto de uma dependência comprometida não se limita ao código malicioso propriamente dito, mas estende-se à aplicação inteira. O malware consegue, por exemplo, exfiltrar dados sensíveis armazenados localmente, modificar funcionalidades da aplicação, comunicar com servidores externos para efeitos de comando e controlo, ou mesmo propagar-se através da instalação de dependências adicionais. Esta dinâmica estabelece um cenário em que a segurança da aplicação deixa de depender apenas da qualidade do seu código-fonte, e passa a estar subordinada à integridade de potencialmente centenas de componentes desenvolvidos por terceiros (Sonatype 2024).

2.1.2 Herança Baseada em Protótipos e Mutação em Tempo de Execução

O modelo de herança de objetos em JavaScript é baseado em protótipos, mecanismo através do qual objetos herdam propriedades de outros objetos por meio de uma cadeia prototípica. Uma característica distintiva deste modelo é que a cadeia de protótipos pode ser modificada em tempo de execução, sem restrições de acesso ou validação (Cornelissen, Shcherbakov e Balliu 2024). Esta capacidade de mutação cria um vetor de ataque conhecido como poluição de protótipos, através do qual um atacante consegue injetar propriedades arbitrárias em protótipos raiz, afetando o comportamento de todos os objetos criados posteriormente na aplicação (Cornelissen, Shcherbakov e Balliu 2024).

O mecanismo de exploração segue um padrão comum em bibliotecas JavaScript. Muitas implementações realizam operações de *merge* de objetos (uma prática arquitetural comum para manipulação de configurações) que não validam devidamente as chaves processadas. Um atacante consegue fornecer dados com propriedades especiais, como `__proto__` ou `constructor.prototype`, que, ao serem processadas pelo código de *merge*, modificam o protótipo raiz (Shcherbakov, Moosbrugger e Balliu 2024). Consequentemente, uma propriedade injetada no *Object.prototype* é herdada por todos os objetos subsequentemente criados,

permitindo que o atacante introduza comportamentos não pretendidos na aplicação sem necessidade de modificar qualquer código da aplicação diretamente (Cornelissen, Shcherbakov e Balliu 2024; Shcherbakov, Moosbrugger e Balliu 2024).

A escala deste problema é evidenciada pelo estudo GHunter, que identificou 56 novos gadgets no Node.js e 67 no Deno, demonstrando que a vulnerabilidade transcende ambientes de execução específicos. Estes gadgets permitem diversos tipos de exploração: 19 casos resultaram em execução arbitrária de código, 31 em escalada de privilégios e 13 em travessia de caminhos (Cornelissen, Shcherbakov e Balliu 2024). Complementarmente, Shcherbakov, Moosbrugger e Balliu (2024) documentaram fluxos de código vulneráveis com potencial de execução remota de código em 49 pacotes populares, incluindo ferramentas amplamente utilizadas como ejs, nodemailer e workerpool (Shcherbakov, Moosbrugger e Balliu 2024). Esta linha de evidência é reforçada pelo estudo Bullseye, que analisou 44.513 pacotes Node.js com pelo menos 10.000 downloads semanais e identificou vulnerabilidades zero-day de poluição de protótipos em 290 pacotes, com provas de conceito comunicadas aos respetivos responsáveis pela manutenção (Houis et al. 2026). O que distingue este vetor de ataque é que a exploração não requer interação direta com funcionalidades sensíveis. Em vez disso, tira partido de propriedades de objetos legítimos que são processadas implicitamente pela aplicação.

O conceito de gadget chain é particularmente relevante para compreender a efetividade deste ataque. Um gadget é uma sequência de operações de código legítimo que, quando encadeadas, realizam um objetivo malicioso (Cornelissen, Shcherbakov e Balliu 2024). Por exemplo, uma biblioteca de processamento de configurações consegue validar propriedades injetadas através de poluição de protótipos, que são subsequentemente utilizadas por um componente de autenticação para determinar privilégios de utilizadores. Neste cenário, nenhum dos componentes tem código malicioso, mas a cadeia de dependências e operações cria um caminho para exploração. Esta característica torna a deteção de poluição de protótipos particularmente desafiadora, uma vez que o código malicioso não é explicitamente presente no código da aplicação.

Embora ambientes de execução modernos como o Deno implementem sistemas de permissões granulares para acesso a recursos, estes permanecem vulneráveis à poluição de protótipos devido à natureza partilhada do motor de execução (AlHamdan e Staicu 2025). O isolamento baseado em WebAssembly é relevante neste contexto porque encapsula código em módulos com memória linear isolada (Haas et al. 2017). Em particular, quando dependências de terceiros são executadas no mesmo contexto JavaScript, uma mutação ao `Object.prototype` pode propagar-se dentro do processo de execução, afetando componentes que não participaram diretamente na introdução da mutação (Cornelissen, Shcherbakov e Balliu 2024; Shcherbakov, Moosbrugger e Balliu 2024).

O isolamento baseado em WebAssembly contribui para mitigar este vetor sobretudo por limitar a propagação da mutação entre fronteiras de componentes/módulos isolados, e não por impedir que a poluição exista. Ao encapsular uma dependência num módulo Wasm, a execução passa a ocorrer sobre uma memória linear privada (Haas et al. 2017) e não sobre o *heap* de objetos do JavaScript, reduzindo a capacidade do componente isolado afetar estruturas globais como protótipos ou estado partilhado do hospedeiro. Consequentemente, mesmo que uma dependência isolada seja explorada, o impacto tende a ficar confinado ao módulo, em vez de se propagar para outras bibliotecas e para a aplicação como um todo, uma propriedade explorada em cenários de *micro-sandboxing* (Narayan et al. 2020). Assim, o contributo do WebAssembly deve ser entendido como um mecanismo de contenção que

restringe a propagação entre dependências e a aplicação hospedeira, mantendo a necessidade de práticas defensivas no código JavaScript que permaneça no mesmo contexto de execução.

2.1.3 Sistema de Módulos Dinâmico e Acesso Irrestrito a Recursos

O terceiro aspeto arquitetural que contribui para a suscetibilidade do npm a ataques da cadeia de abastecimento é a natureza dinâmica do sistema de resolução de módulos. Em JavaScript, a função `require()` aceita argumentos que podem ser computados em tempo de execução, o que significa que a resolução de qual módulo será carregado é determinada apenas durante a execução do código. Esta característica impossibilita análise estática abrangente, uma vez que ferramentas de segurança não conseguem antecipadamente determinar que módulos vão ser carregados (Staicu, Pradel e Livshits 2018). Um estudo de 235.850 módulos Node.js revelou vulnerabilidades de injeção generalizadas neste contexto, onde entradas não validadas podem ser utilizadas para manipular caminhos de módulos e carregar código não pretendido (Staicu, Pradel e Livshits 2018).

Relacionado com esta característica encontra-se o problema das dependências transitivas. Quando um desenvolvedor executa `npm install package a`, o gestor de pacotes instala não apenas o pacote `a`, mas também todas as suas dependências (Zimmermann et al. 2019). Dados atuais do ecossistema indicam que cerca de 64% dos componentes desenvolvidos por terceiros presentes nas aplicações modernas são dependências transitivas, sendo a maioria quase impossível de localizar sem recorrer a uma ferramenta automatizada (BlackDuck 2025). Esta invisibilidade estrutural reduz significativamente o conhecimento do desenvolvedor sobre a superfície de ataque da sua aplicação, uma vez que pacotes que nem foram explicitamente solicitados nem definidos no ficheiro `package.json` são introduzidos no ambiente (Zahan et al. 2022; Zimmermann et al. 2019). Um atacante necessita apenas de comprometer uma dependência amplamente utilizada para que o malware seja distribuído a milhares de aplicações através da cadeia de abastecimento (Zahan et al. 2022; Zimmermann et al. 2019).

Para além da dinâmica do sistema de módulos, os scripts de instalação executam com privilégios totais do sistema (Staicu, Pradel e Livshits 2018; Zimmermann et al. 2019). Estes scripts executam durante a fase de instalação, antes de qualquer código da aplicação ser invocado, e conseguem aceder sem restrições ao sistema de ficheiros, rede e processos do sistema operativo (Rayarao e Donikena 2025; Zahan et al. 2022). Esta característica permite que malware implantado em scripts de instalação execute com privilégios totais e de forma invisível para o desenvolvedor. A capacidade de executar código arbitrário nesta fase, combinada com a invisibilidade das dependências transitivas, cria um vetor de ataque robusto que contorna muitos mecanismos convencionais de deteção de vulnerabilidades (Rayarao e Donikena 2025; Sonatype 2024; Zahan et al. 2022).

A Tabela 2.2 resume as dimensões estruturais analisadas nesta secção, os mecanismos que as tornam exploráveis, as respetivas consequências e a razão pela qual os mecanismos atuais do ecossistema não eliminam estas vulnerabilidades.

Tabela 2.2: Vulnerabilidades estruturais do ecossistema JavaScript/npm

Dimensão	Mecanismo	Consequência	Limite dos mecanismos atuais
Privilégios totais	Ausência de um modelo nativo de capacidades; bibliotecas importadas herdam os privilégios da aplicação.	Código de terceiros pode aceder a APIs globais e recursos sensíveis sem restrições granulares.	<i>Audit</i> , ficheiros de bloqueio e proveniência validam artefactos, mas não reduzem autoridade em execução.
Poluição de protótipos	Mutabilidade da cadeia de protótipos e processamento inseguro de propriedades como <code>__proto__</code> ou <code>constructor.prototype</code> .	Propriedades injetadas podem afetar objetos em toda a aplicação e viabilizar cadeias de exploração.	Os mecanismos JavaScript/npm não isolam protótipos partilhados nem cobrem todos os comportamentos sem vulnerabilidade conhecida.
Módulos dinâmicos e dependências transitivas	Resolução dinâmica de módulos, scripts de instalação e introdução indireta de pacotes.	A superfície real de ataque torna-se difícil de auditar e pode executar código com privilégios elevados.	Instalação reproduzível e auditoria não tornam o grafo de dependências benigno nem eliminam código indireto privilegiado.

2.1.4 Mecanismos de Mitigação no Ecossistema npm e seus Limites

O ecossistema npm dispõe de mecanismos de mitigação que atuam em diferentes pontos da cadeia de abastecimento. No momento da instalação, o ficheiro *package-lock.json* descreve uma representação concreta da árvore de dependências, reduzindo variações entre instalações causadas por resolução dinâmica de versões, diferenças entre versões do gestor de pacotes ou novas publicações compatíveis com intervalos semânticos (*package-lock.json* / *npm Docs* 2026). O comando *npm audit*, por sua vez, envia uma descrição das dependências configuradas no projeto para o registo e solicita um relatório de vulnerabilidades conhecidas, incluindo informação sobre pacote afetado, severidade, caminho na árvore de dependências e, quando disponível, sugestões de correção (*Auditing package dependencies for security vulnerabilities* / *npm Docs* 2026; *npm-audit* / *npm Docs* 2026).

No processo de publicação, o npm combina mecanismos de autenticação e verificação de origem. A publicação de pacotes exige autenticação de dois fatores ou tokens granulares configurados para não exigir esse mecanismo, e as definições do pacote podem impor autenticação de dois fatores e impedir o uso de tokens de publicação (*Requiring 2FA for package publishing and settings modification* / *npm Docs* 2026). Em paralelo, o *trusted publishing* permite publicar pacotes diretamente a partir de fluxos CI/CD usando OpenID Connect (OIDC), substituindo tokens npm de longa duração por credenciais curtas e associadas ao fluxo autorizado (*Trusted publishing for npm packages* / *npm Docs* 2026). Quando usado

com GitHub Actions ou GitLab CI/CD em repositórios e pacotes públicos, este mecanismo gera automaticamente atestações de proveniência que associam o pacote ao ambiente de construção, ao repositório, ao *commit* e ao ficheiro de *workflow* usados na publicação (*Trusted publishing for npm packages* | *npm Docs* 2026; *Viewing package provenance* | *npm Docs* 2026).

O npm também disponibiliza mecanismos de verificação no lado do consumidor. O comando *npm audit signatures* verifica assinaturas ECDSA do registo e atestações de proveniência associadas aos pacotes descarregados, reportando erro quando encontra assinaturas ou atestações ausentes ou inválidas num registo que suporta estes mecanismos (*npm-audit* | *npm Docs* 2026; *Verifying ECDSA registry signatures* | *npm Docs* 2026). Estes mecanismos reforçam a integridade, a autenticidade e a rastreabilidade dos artefactos, e complementam modelos como Supply-chain Levels for Software Artifacts (SLSA), nos quais a proveniência é usada para verificar se um artefacto corresponde às expectativas definidas para a sua origem e processo de construção (*Verifying artifacts* 2026).

Apesar destas melhorias, estes mecanismos não eliminam o problema de autoridade em tempo de execução. O *npm audit* depende da existência de vulnerabilidades conhecidas e reportadas (*Auditing package dependencies for security vulnerabilities* | *npm Docs* 2026). Os ficheiros de bloqueio tornam a instalação mais reproduzível, mas não determinam se o código fixado é benigno (*package-lock.json* | *npm Docs* 2026). Por último, a proveniência comprova onde e como um artefacto foi produzido, não que o seu comportamento seja seguro nos contextos de execução (*Verifying artifacts* 2026).

Assim, as defesas do npm reduzem riscos de resolução, publicação, autenticação, auditoria e verificação de artefactos, mas não impõem, por si só, uma fronteira granular entre a aplicação hospedeira e uma dependência em execução. Uma dependência publicada de forma válida, instalada de forma reproduzível e sem vulnerabilidades conhecidas continua a executar segundo o modelo de autoridade do ambiente JavaScript que a consome. Por essa razão, estes mecanismos devem ser entendidos como complementares a técnicas de isolamento em tempo de execução, e não como substitutos.

2.2 WebAssembly: Fundamentos e Características

Esta secção introduz o Wasm como tecnologia relevante para compreender abordagens modernas de execução portátil e isolamento, explorando as características que o distinguem dos ambientes de execução tradicionais. A subsecção 2.2.1 descreve a arquitetura da máquina virtual baseada em pilha e o seu modelo de memória linear, elementos que suportam a segurança da memória e o isolamento do fluxo de controlo por conceção. Na subsecção 2.2.2, analisa-se a viabilidade do Wasm em ambientes de produção através de estudos comparativos de desempenho e eficiência energética face ao JavaScript. Por último, a subsecção 2.2.3 detalha o Modelo de Componentes, uma evolução recente da especificação que facilita a comunicação entre módulos desenvolvidos com linguagens distintas.

2.2.1 Arquitetura e Modelo de Execução

A arquitetura do WebAssembly foi desenhada para permitir a execução segura e eficiente de código de baixo nível na Web. Conforme estabelecido por Haas et al. (2017), o Wasm define-se formalmente como uma máquina virtual baseada em pilha (Haas et al. 2017). Esta opção de desenho, em detrimento de uma arquitetura baseada em registos, permite

uma codificação binária compacta, uma vez que as instruções não necessitam de listar explicitamente os operandos, minimizando o tempo de transferência pela rede (Haas et al. 2017; Kim, Jang e Shin 2022).

A unidade fundamental de distribuição é o módulo, que encapsula o código e os dados necessários. A estrutura do módulo, ilustrada na Figura 2.1, é organizada em secções distintas (Kim, Jang e Shin 2022). Esta organização facilita a descodificação paralela e o streaming compilation, permitindo ao browser iniciar a geração de código máquina durante o descarregamento do ficheiro (Haas et al. 2017).

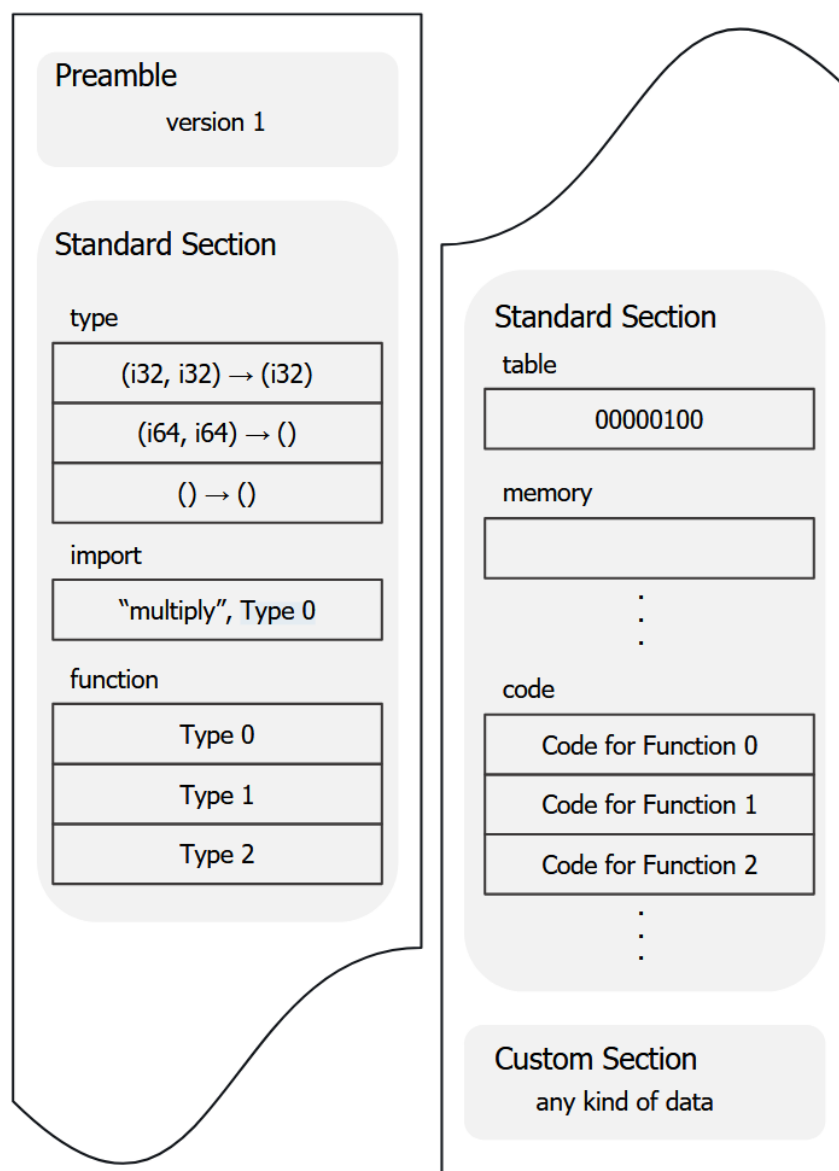


Figura 2.1: Módulo WebAssembly (Kim, Jang e Shin 2022)

Um componente central da arquitetura é o sistema de Fluxo de Controlo Estruturado. Ao contrário de representações intermédias que permitem saltos arbitrários, o Wasm restringe o fluxo de controlo a construções estruturadas. Esta restrição permite a validação do código numa única passagem, assegurando a verificação de tipos e a integridade da memória em tempo linear durante o carregamento (Haas et al. 2017; Kim, Jang e Shin 2022).

O modelo de memória baseia-se numa Memória Linear, constituída por um array de bytes contíguo, isolado do espaço de endereçamento do processo hospedeiro (Haas et al. 2017; WebAssembly 2025). O acesso a esta memória é confinado, sendo que qualquer tentativa de acesso fora dos limites do array provoca uma interrupção imediata. Adicionalmente, o Wasm separa o código dos dados, impedindo a injeção de código executável na memória de dados e mitigando vetores de ataque comuns (Haas et al. 2017; Kim, Jang e Shin 2022). A Figura 2.2 clarifica este modelo de memória em cada módulo.

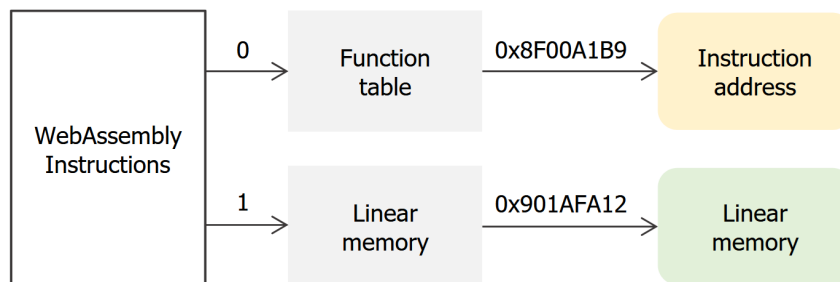


Figura 2.2: Modelo de Memória Linear de um módulo WebAssembly (Kim, Jang e Shin 2022)

A evolução recente da especificação alarga este modelo base sem alterar os seus princípios fundamentais. A versão WebAssembly 3.0 adicionou funcionalidades como *Memory64*, múltiplas memórias por módulo, garbage collection, referências tipadas, chamadas em cauda (*tail calls*) e tratamento de exceções, melhorando o suporte à compilação de linguagens de alto nível e a modelos de execução mais complexos sobre Wasm (*Wasm 3.0 Completed - WebAssembly 2025*).

2.2.2 Desempenho e Eficiência

O desempenho do WebAssembly decorre da sua representação binária e do modelo de validação, que aceleram o processo de descodificação e compilação inicial pelo motor do browser (Haas et al. 2017). Ao contrário do JavaScript, que depende de compilação Just-In-Time (JIT) e otimizações especulativas, o Wasm oferece um comportamento de execução previsível.

No que concerne ao tempo de execução, estudos empíricos indicam uma vantagem do WebAssembly. De Macedo et al. (2022) conduziram uma análise sistemática comparando a execução de algoritmos e aplicações em diversos browsers, verificando que o WebAssembly supera o JavaScript na maioria dos cenários de teste mais complexos em ambientes reais, com uma vantagem média de 17,24% (De Macedo et al. 2022). Já no caso de *micro-benchmarks*, o JavaScript consegue superar o WebAssembly nalguns casos devido às otimizações realizadas ao longo do tempo pelo compilador JIT, que pode conseguir atingir um desempenho próximo ao nativo. (De Macedo et al. 2022).

Esta diferença no tempo de execução reflete-se na eficiência energética. O mesmo estudo de De Macedo et al. (2022) quantificou que a utilização de WebAssembly permitiu uma redução média de 30% no consumo de energia face a implementações equivalentes em JavaScript (De Macedo et al. 2022). Este dado é relevante para dispositivos móveis, onde a eficiência do processamento influencia a autonomia do equipamento.

Para além do ambiente do browser, a eficiência do WebAssembly observa-se em contextos de Cloud e Edge Computing. A reduzida dimensão dos módulos Wasm resulta em tempos de arranque inferiores aos de contentores tradicionais. Besozzi et al. (2025) notam que, em cenários de computação de ponta, o WebAssembly oferece uma latência de inicialização reduzida, adequando-se a funções *serverless*, embora o desempenho possa variar em cargas de trabalho com Input/Output (I/O) intensivo (Besozzi et al. 2025).

2.2.3 Component Model

O módulo WebAssembly, na sua conceção original, apresenta limitações na composição e comunicação entre módulos desenvolvidos em linguagens distintas. As funções restritas à utilização de tipos numéricos inteiros ou de ponto flutuante exige a implementação de código de adaptação complexo para a tradução de estruturas de dados compostas utilizadas em linguagens de alto nível. Para mitigar este problema de interoperabilidade e estruturar o suporte multilinguagem, a Bytecode Alliance propôs a arquitetura do Modelo de Componentes (Alliance 2025d).

Esta arquitetura visa a construção de sistemas modulares através da definição de interfaces agnósticas à linguagem de implementação usando a linguagem WebAssembly Interface Types (WIT) (Alliance 2025a). O modelo introduz o conceito de Componente como uma unidade de implantação que encapsula módulos WebAssembly e define explicitamente as suas dependências e capacidades através de Interfaces e Mundos (Alliance 2025b). As Interfaces usam WIT para definir contratos a partir dos quais os Componentes podem interagir entre si e com os hospedeiros, sendo as representações de baixo nível definidas através da Canonical Application Binary Interface (ABI) (Alliance 2025d). Cada Interface agrupa tipos e funções relacionados que, em conjunto, definem uma unidade delimitada de funcionalidade, podendo ainda ser composta através de importação e exportação (Alliance 2025c). Já os Mundos descrevem os Componentes através da especificação das Interfaces que estes exportam e importam (Alliance 2025e). Este mecanismo abstrai a gestão de memória e a serialização de dados, permitindo a comunicação direta entre componentes desenvolvidos em linguagens de alto nível distintas (Alliance 2025e).

A interação com linguagens dinâmicas, como JavaScript, é padronizada através deste modelo. O sistema define uma camada de tradução automática que mapeia os tipos complexos do Componente definidos com WIT para os objetos do ambiente hospedeiro, reduzindo a fricção causada pela incompatibilidade entre a memória linear do WebAssembly e o Garbage Collector do JavaScript. A especificação assegura que esta interoperabilidade ocorre sem comprometer o isolamento de memória, garantindo que a falha ou comprometimento de um componente não afeta diretamente a integridade dos restantes (Alliance 2025e).

A Figura 2.3 esquematiza esta arquitetura de composição. Ao contrário de bibliotecas partilhadas tradicionais, os componentes mantêm fronteiras de isolamento estritas, sendo a comunicação mediada pela plataforma através de interfaces WIT.

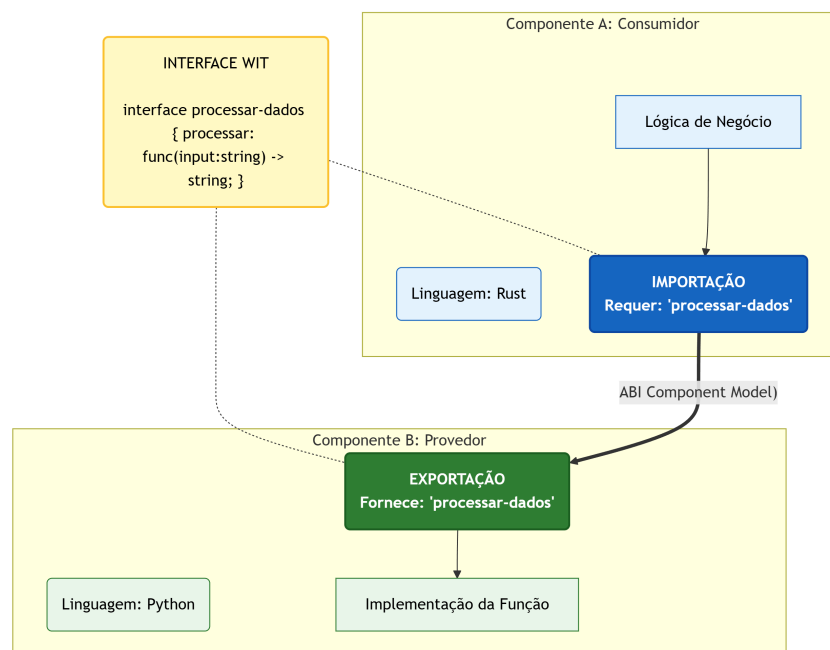


Figura 2.3: Ilustração conceitual do Modelo de Componentes WebAssembly. O diagrama demonstra como dois componentes isolados, desenvolvidos em linguagens distintas (Rust e Python), comunicam através de uma interface WIT.

Capítulo 3

Estado da Arte

O presente capítulo apresenta a revisão do estado da arte relevante para a solução proposta, partindo do enquadramento estabelecido no capítulo anterior. O objetivo é analisar a literatura e as alternativas técnicas que ajudam a posicionar uma abordagem baseada em WebAssembly para isolamento granular de bibliotecas JavaScript/npm.

Inicialmente, a Secção 3.1 detalha o processo metodológico aplicado na seleção e análise das fontes bibliográficas, garantindo a transparência e a reprodutibilidade do processo de revisão.

De seguida, a Secção 3.2 analisa o problema específico da integração entre JavaScript e WebAssembly. Esta secção compara as duas alternativas principais consideradas para executar ou expor código JavaScript através de uma fronteira Wasm: a abordagem baseada em *jco* e no Modelo de Componentes, e a abordagem baseada em QuickJS e bibliotecas relacionadas. A comparação evidencia os desafios de modelação de interfaces, conversão de dados, preservação da semântica JavaScript e exposição controlada de capacidades do hospedeiro.

Para contextualizar a solução no panorama atual, a Secção 3.3 apresenta estratégias de isolamento existentes, desde a virtualização tradicional e contentores até às abordagens mais modernas baseadas em processos leves. Esta revisão permite compreender de forma mais clara os compromissos entre segurança e desempenho que as soluções atuais enfrentam.

Por fim, na Secção 3.4, cruzam-se os conhecimentos anteriores para realizar uma análise comparativa direta entre o WebAssembly e outras tecnologias de isolamento, incluindo V8 Isolates, Cage4Deno e JSand. Esta secção evidencia as lacunas que permanecem em aberto e clarifica os compromissos entre isolamento, desempenho, controlo de acesso e granularidade.

3.1 Método de Pesquisa

Esta secção estabelece o rigor metodológico que suporta a revisão da literatura, descrevendo o protocolo de Mapeamento Sistemático (*Systematic Mapping Review*) aplicado. O processo de investigação organiza-se em duas vertentes distintas: a exploração do espaço do problema, focado nas vulnerabilidades da cadeia de abastecimento JavaScript/npm, e a investigação do espaço da solução, centrada nos mecanismos de segurança do WebAssembly e técnicas de isolamento concorrentes. O texto detalha as estratégias adotadas para cada dimensão, discriminando as palavras-chave e os critérios de filtragem que orientaram a seleção dos estudos. Adicionalmente, fundamenta-se a utilização de métodos complementares, como o *snowballing* e a análise de literatura técnica, para assegurar a cobertura de

desenvolvimentos recentes. A transparência e reprodutibilidade do processo de seleção são evidenciadas através da inclusão de diagramas PRISMA.

Para o mapeamento do espaço do problema, aplicou-se a *string* de pesquisa: `(( "JavaScript" OR "npm" ) AND ( "security" OR "vulnerability" OR "attack" ) AND ( "ambient authority" OR "least privilege" OR "third-party code" OR "untrusted code" OR "supply chain" OR "malicious package" ))`. Aplicou-se ainda um filtro temporal para estudos publicados a partir de 2010, circunscrevendo a análise ao período de expansão do ecossistema npm e consequente crescimento dos ataques à cadeia de abastecimento. Este processo, ilustrado na Figura 3.1, resultou na seleção de 7 estudos que caracterizam as vulnerabilidades arquiteturais e a tipologia de ataques.

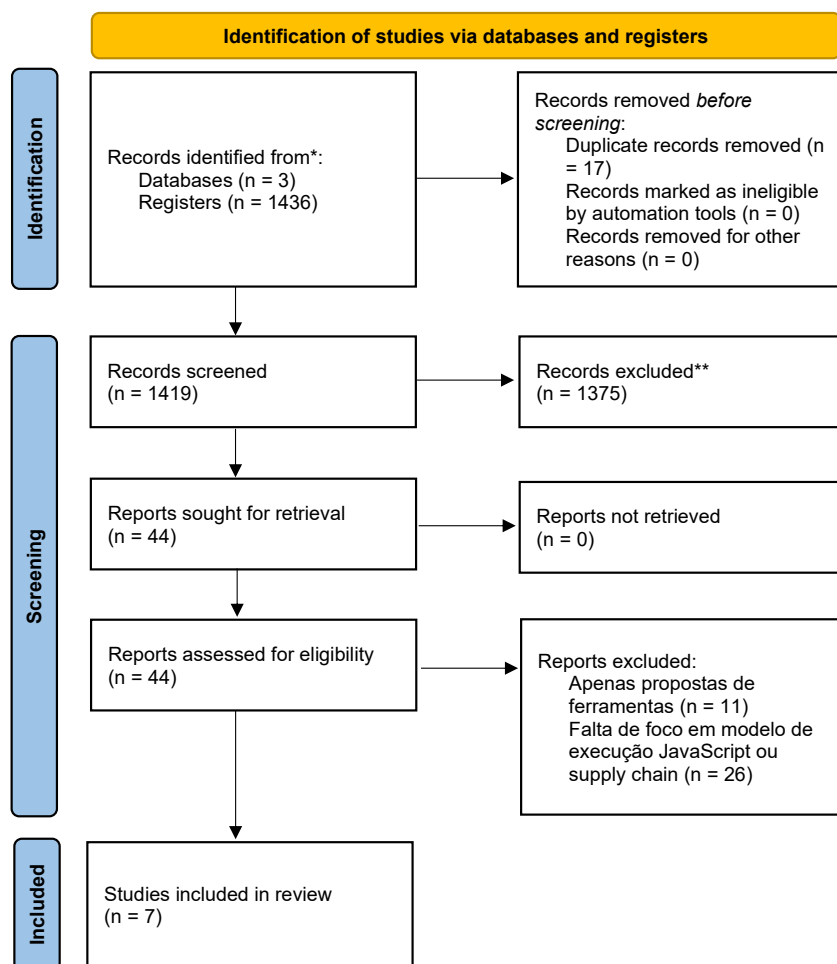


Figura 3.1: Diagrama PRISMA para a pesquisa sobre o ecossistema JavaScript/npm

Para o espaço da solução, utilizou-se a string `("WebAssembly" OR "Wasm" ) AND ( "security" OR "isolation" OR "sandboxing" OR "memory safety" )`. A triagem deste conjunto, detalhada na Figura 3.2, permitiu identificar 6 estudos focados em Wasm. Aplicou-se um filtro temporal para trabalhos publicados a partir de 2017, ano de publicação do artigo que introduziu formalmente o WebAssembly (Haas et al. 2017), garantindo a exclusão de resultados não relacionados com a tecnologia.

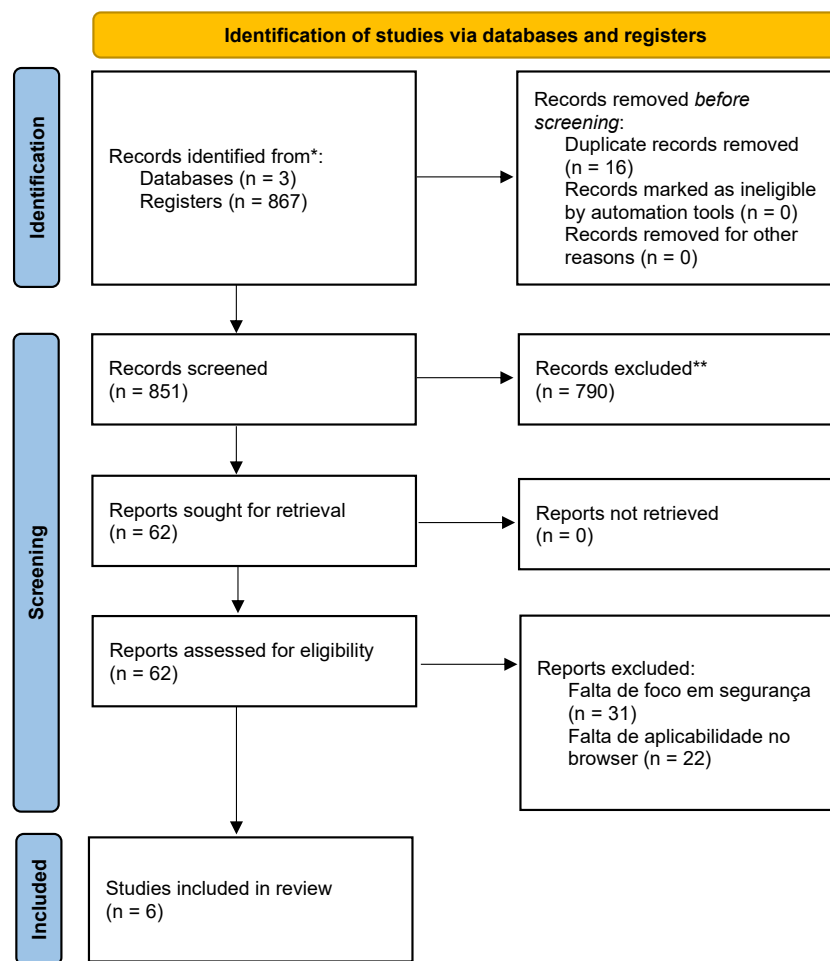


Figura 3.2: Diagrama PRISMA para a pesquisa sobre WebAssembly e isolamento

Para complementar a pesquisa sistemática, foram aplicados métodos adicionais. Através de *Snowballing* a partir dos estudos primários, identificaram-se 5 artigos adicionais relevantes, maioritariamente relacionados com técnicas de isolamento. Por fim, foram incluídas referências de documentação técnica oficial e relatórios da indústria, garantindo o acesso a especificações, ferramentas e dados de segurança mais recentes não disponíveis em canais puramente académicos.

3.2 Integração entre JavaScript e WebAssembly

A utilização do WebAssembly como mecanismo de isolamento para bibliotecas JavaScript introduz um problema adicional de interoperabilidade. O ecossistema npm é composto por bibliotecas com interfaces frequentemente dinâmicas, baseadas em objetos mutáveis, propriedades opcionais, callbacks, classes, exceções e mecanismos assíncronos. Por outro lado, o WebAssembly opera sobre uma memória linear isolada e, no caso do Modelo de Componentes, favorece interfaces explícitas e tipadas. Assim, uma solução de isolamento baseada em Wasm precisa de resolver duas questões complementares: executar código JavaScript dentro de um módulo Wasm, e preservar a comunicação entre o hospedeiro e o código isolado sem destruir a semântica dos dados.

Esta segunda questão é particularmente relevante para bibliotecas JavaScript. No sentido do hospedeiro para a Máquina Virtual (VM), a aplicação precisa de invocar funções isoladas e passar argumentos, objetos de configuração ou callbacks. Esses valores têm de permanecer utilizáveis do lado isolado, mesmo quando atravessam uma fronteira entre o *heap* JavaScript do hospedeiro e a memória do módulo Wasm. No sentido inverso, da VM para o hospedeiro, os resultados das chamadas, exceções, objetos retornados, funções ou valores assíncronos precisam de ser convertidos de forma a continuar utilizáveis pela aplicação. A compatibilidade de uma biblioteca isolada depende, portanto, não apenas das capacidades do hospedeiro que são expostas, mas também da fidelidade da conversão de dados nessa fronteira.

3.2.1 jco

O projeto *jco*, desenvolvido pela Bytecode Alliance, constitui uma ferramenta para trabalhar com componentes WebAssembly em ambientes JavaScript (Alliance 2026b). Entre as suas funcionalidades incluem-se a transpilação de WebAssembly Components para módulos ECMAScript, o suporte a WASI Preview 2 em ambientes como Node.js e browsers, e a possibilidade de criar componentes a partir de código JavaScript através de *ComponentizeJS* (Alliance 2026b). Esta abordagem está alinhada com a visão do Modelo de Componentes, em que as fronteiras entre componentes são descritas através de interfaces WIT e em que a comunicação é mediada por contratos explícitos (Alliance 2025a,c).

Do ponto de vista conceptual, esta estratégia é atrativa porque aproxima JavaScript do ecossistema de componentes WebAssembly. Um componente passa a poder declarar as funções que exporta e as capacidades que importa, permitindo que a ligação com o hospedeiro seja descrita de forma explícita. No entanto, esta mesma característica torna-se uma limitação importante quando o objetivo é isolar bibliotecas JavaScript arbitrárias. As APIs JavaScript raramente se reduzem a assinaturas estáticas simples: muitas expõem objetos com propriedades dinâmicas, callbacks, valores com vários tipos possíveis, *promises*, classes, exceções e estruturas mutáveis. A tradução destas interfaces para WIT pode tornar-se complexa tanto manualmente como de forma automática, porque vários destes padrões não têm uma correspondência direta com os tipos e contratos esperados pelo Modelo de Componentes.

Esta dificuldade é agravada quando a biblioteca isolada precisa de interagir com capacidades do hospedeiro, como APIs Web. Muitas APIs do browser são compostas por objetos com identidade, estado interno, métodos, eventos e restrições de invocação. Representar essas capacidades como importações WIT exige modelar explicitamente uma parte substancial da superfície do hospedeiro, incluindo os tipos de entrada e saída esperados por cada operação. Para o caso de uso desta dissertação, em que se pretende controlar seletivamente o acesso de bibliotecas JavaScript a capacidades do ambiente, esta necessidade de modelação prévia torna a abordagem pouco prática.

Além disso, a documentação do *jco* classifica o projeto como experimental e sem garantias de estabilidade, segurança ou suporte (Alliance 2026b). O *ComponentizeJS*, por sua vez, recorre a uma incorporação de um motor SpiderMonkey para executar código JavaScript dentro de um componente, o que introduz um custo adicional de tamanho e complexidade (Alliance 2026a). Assim, embora o *jco* represente uma direção relevante para componentes WebAssembly com interfaces bem definidas, a combinação entre interfaces WIT, APIs JavaScript dinâmicas e exposição de capacidades Web torna esta alternativa pouco adequada para o isolamento transparente de bibliotecas JavaScript existentes.

3.2.2 QuickJS e Bibliotecas Relacionadas

Uma alternativa distinta consiste em executar JavaScript dentro de um motor JavaScript embutido e compilar esse motor para WebAssembly. O QuickJS é um motor JavaScript pequeno e embutível, implementado em C, com suporte para funcionalidades modernas da linguagem, incluindo módulos, *async/await*, geradores, *proxies* e *BigInt* (Bellard e Gordon 2026a). A sua API distingue runtimes e contextos: um *runtime* contém o estado de execução e a gestão de memória, enquanto um contexto representa um ambiente global onde o código JavaScript é avaliado. A documentação descreve ainda mecanismos de controlo como limites de memória, limites de pilha e funções de interrupção da execução, características relevantes para cenários em que código não confiável é executado num ambiente controlado (Bellard e Gordon 2026b).

Quando compilado para WebAssembly, o QuickJS permite que a biblioteca JavaScript seja executada por um interpretador contido dentro da fronteira Wasm. Esta abordagem evita a necessidade de traduzir previamente a API da biblioteca para uma interface WIT e preserva melhor a semântica dinâmica do JavaScript. Em vez de transformar cada biblioteca num componente com um contrato estático, executa-se o próprio código JavaScript num *runtime* separado, ficando o principal desafio deslocado para a fronteira entre o hospedeiro e a VM: que capacidades são expostas, que valores atravessam a fronteira e como esses valores são convertidos.

A biblioteca *quickjs-emsripten* fornece precisamente esta camada prática de integração. O projeto disponibiliza bindings JavaScript e TypeScript para uma versão do QuickJS compilada para WebAssembly, permitindo criar programaticamente módulos, runtimes e contextos QuickJS a partir do hospedeiro JavaScript (Lazaroff 2026). A biblioteca permite avaliar código dentro da VM, manipular valores através de *handles*, expor funções do hospedeiro ao ambiente isolado e configurar aspetos como limites de memória ou interrupção de execução. Também disponibiliza variantes síncronas e assíncronas, incluindo suporte baseado em Asyncify para cenários em que código executado na VM precisa de interagir com operações assíncronas do hospedeiro (Lazaroff 2026).

Esta camada reduz parte do trabalho necessário para criar uma *sandbox* JavaScript sobre WebAssembly, mas não elimina o problema de interoperabilidade. Os valores que existem dentro da VM são representados por *handles*, cuja gestão de ciclo de vida tem de ser controlada explicitamente para evitar fugas de memória. Além disso, objetos complexos, funções, erros, buffers, datas ou estruturas mutáveis exigem conversões cuidadosas para preservar comportamento observável nos dois lados da fronteira. A compatibilidade com bibliotecas JavaScript depende, portanto, da capacidade de construir uma membrana que consiga transportar argumentos para a VM e devolver resultados para o hospedeiro com fidelidade suficiente.

O *quickjs-emsripten-sync* surge como uma camada complementar a este problema. A biblioteca introduz a abstração de *Arena*, que facilita a gestão de *handles*, a avaliação de código, a exposição de objetos do hospedeiro e a sincronização de valores entre o ambiente JavaScript hospedeiro e o contexto QuickJS (Re:Earth 2026). Em particular, recorre a mecanismos de *marshalling*, *unmarshalling* e *proxies* para tornar objetos e funções mais utilizáveis através da fronteira de isolamento. Esta característica é relevante para a exposição controlada de capacidades do hospedeiro, como APIs Web, porque permite fornecer ao código isolado referências mediadas em vez de acesso direto ao ambiente global.

Outro aspeto relevante é a possibilidade de extensão através de *marshallers* personalizados (Re:Earth 2026). Esta capacidade permite adaptar a conversão de tipos aos padrões usados por bibliotecas específicas, aumentando a compatibilidade sem alterar o motor QuickJS ou a fronteira Wasm subjacente. Por exemplo, bibliotecas que dependem de tipos ou objetos não suportados diretamente pela conversão padrão podem ser acomodadas através de regras específicas de transformação. Ainda assim, esta flexibilidade tem custos: quanto maior for o número de objetos e capacidades expostos à VM, maior é a superfície que a membrana precisa de controlar, e conversões incorretas podem quebrar compatibilidade ou introduzir comportamentos inesperados.

Em síntese, o *jco* oferece uma abordagem alinhada com o Modelo de Componentes, mas a dependência de interfaces WIT torna-se um obstáculo quando o alvo são bibliotecas JavaScript com APIs dinâmicas e necessidade de interação com APIs Web. A abordagem baseada em QuickJS desloca o problema: em vez de modelar antecipadamente a interface completa da biblioteca, executa JavaScript real dentro de um motor isolado em WebAssembly. As bibliotecas *quickjs-emsripten* e *quickjs-emsripten-sync* tornam esta estratégia programaticamente viável ao fornecer criação de runtimes, avaliação de código, exposição controlada de capacidades, conversão de valores e extensão por *marshallers* personalizados. Esta comparação estabelece a base técnica para a discussão de conceção apresentada nos capítulos seguintes.

A Tabela 3.1 sintetiza a comparação entre as duas alternativas principais consideradas para integrar JavaScript e WebAssembly no contexto deste trabalho.

Tabela 3.1: Comparação entre *jco* e QuickJS para integração JavaScript-WebAssembly

Critério	<i>jco</i> / Component Model	QuickJS em WebAssembly
Modelo de integração	Componentes WebAssembly com interfaces explícitas definidas em WIT.	Motor JavaScript embutido e compilado para WebAssembly, executando código JavaScript dentro da VM.
Adequação a bibliotecas JavaScript	Adequado para interfaces estáveis e tipadas, mas limitado perante APIs JavaScript dinâmicas.	Mais adequado para bibliotecas JavaScript existentes, pois preserva melhor a semântica da linguagem.
Exposição de capacidades do hospedeiro	Requer modelação explícita das capacidades importadas através de contratos WIT.	Permite exposição mediada de capacidades através de <i>proxies</i> e membranas.
Conversão de dados	Baseada nos tipos suportados pelo Modelo de Componentes e pela Canonical ABI.	Baseada em <i>handles</i> , <i>marshalling</i> , <i>unmarshalling</i> e <i>marshallers</i> personalizados.
Principal limitação	Necessidade de traduzir APIs JavaScript dinâmicas para interfaces estáticas.	Complexidade da gestão da fronteira entre o hospedeiro JavaScript e a VM QuickJS.

3.3 Técnicas de Isolamento de Software

Esta secção mapeia as diferentes tecnologias de isolamento de software, avaliando a eficácia e os custos operacionais das diferentes estratégias disponíveis atualmente. A análise inicia-se na subsecção 3.3.1, que cobre as abordagens tradicionais baseadas em virtualização e

containerização, discutindo por que razão, apesar da sua robustez no lado do servidor, se revelam impraticáveis para isolamento granular devido ao elevado consumo de recursos. Na subsecção 3.3.2, explora-se o estado da arte do isolamento suportado especificamente por WebAssembly, analisando como esta tecnologia está a ser reaproveitada para criar *sandboxes* leves. Por fim, a subsecção 3.3.3 foca-se especificamente nos ambientes de execução JavaScript, contrastando as soluções disponíveis para o lado do servidor com as restrições únicas do lado do cliente, onde se discutem soluções existentes que oferecem garantias de segurança insuficientes para mitigar ataques à cadeia de abastecimento.

3.3.1 Isolamento Baseado em Processos e Contentores

O isolamento de software constitui um requisito fundamental para a segurança e estabilidade de sistemas computacionais, especialmente em ambientes de computação em nuvem onde a infraestrutura física é partilhada por múltiplos locatários (Wiegratz 2024). Historicamente, este isolamento tem sido assegurado através de mecanismos implementados ao nível do sistema operativo ou mediante a virtualização de hardware.

O mecanismo mais elementar de segregação num sistema operativo é o processo. Num sistema de tempo partilhado, cada processo opera num espaço de endereçamento virtual distinto, impedindo o acesso não autorizado ou a corrupção da memória alocada a outros processos. Contudo, o isolamento puramente baseado em processos é frequentemente insuficiente para a execução de código não confiável, uma vez que a superfície de ataque exposta através da interface de chamadas de sistema permanece ampla e partilhada com o núcleo (Wiegratz 2024).

A evolução para a virtualização ao nível do sistema operativo estende o modelo de processos para criar ambientes de execução segregados designados por contentores. Ao contrário da virtualização baseada em hipervisor, que emula o hardware integralmente para cada VM, os contentores partilham o núcleo do sistema operativo hospedeiro (Wiegratz 2024). O isolamento lógico neste modelo é concretizado principalmente através de duas primitivas do núcleo Linux: espaços de nomes e Control Groups (cgroups). (Wiegratz 2024).

Os espaços de nomes proporcionam o isolamento de visualização, garantindo que um grupo de processos aceda apenas aos recursos globais do sistema que lhe foram explicitamente atribuídos, tais como a árvore de processos, interfaces de rede e pontos de montagem. Complementarmente, os cgroups asseguram o isolamento de recursos, permitindo a limitação e a contabilização do uso de processador e memória, impedindo que um contentor monopolize os recursos do hospedeiro (Wiegratz 2024).

Esta abordagem de isolamento proporciona eficiência no consumo de recursos, permitindo tempos de inicialização reduzidos e uma densidade de implantação superior à das máquinas virtuais. No entanto, a partilha do núcleo do sistema operativo introduz riscos de segurança significativos. Wiegratz destaca que uma vulnerabilidade no núcleo pode comprometer o sistema hospedeiro na sua totalidade, permitindo que processos escapem ao isolamento do contentor e obtenham privilégios elevados no hospedeiro (Wiegratz 2024). Adicionalmente, vulnerabilidades na configuração destes mecanismos, como contentores executados com privilégios excessivos ou com montagens de volumes sensíveis do hospedeiro, aumentam a superfície de ataque disponível para injeção de código ou elevação de privilégios (Kim, Jang e Shin 2022; Wiegratz 2024).

3.3.2 Isolamento com WebAssembly

Enquanto a secção anterior explorou o isolamento ao nível do sistema operativo, o WebAssembly introduz um paradigma de isolamento intra-processo. Esta abordagem permite a execução de código não confiável dentro do mesmo espaço de endereçamento da aplicação hospedeira, mas com garantias fortes de separação de memória e fluxo de controlo (Haas et al. 2017). Ao contrário dos contentores, que dependem do núcleo do sistema operativo para impor barreiras, o Wasm consegue isolamento intra-processo assegurado pelo próprio ambiente de execução através de uma abstração de memória linear isolada. Neste modelo, é assegurado que todos os acessos à memória são limitados aos limites do módulo, impedindo a corrupção do espaço de endereçamento da aplicação hospedeira (Haas et al. 2017).

A eficácia deste modelo reside na granularidade fina do isolamento. Narayan et al. (2020) demonstraram a aplicação prática desta técnica no browser Firefox através do projeto RLBox, que utiliza Wasm para isolar bibliotecas de terceiros dentro do processo de renderização. Esta arquitetura de *micro-sandboxing* confina falhas de segurança e vulnerabilidades de memória da biblioteca isolada, impedindo que estas comprometam o processo pai, com um impacto de desempenho negligenciável para a maioria das cargas de trabalho (Narayan et al. 2020). Este caso de uso ilustra como o Wasm permite converter bibliotecas legadas inseguras em componentes isolados seguros, sem a necessidade de reescrever o código original.

Apesar da robustez teórica, a implementação prática enfrenta desafios. A superfície de ataque desloca-se do kernel para o motor de execução Wasm e para as interfaces de sistema expostas, que devem ser rigorosamente controladas para evitar fugas de isolamento (Kim, Jang e Shin 2022). Recentemente, propostas como o WasmSepa sugerem reforçar este modelo através da separação de privilégios, movendo a execução de componentes críticos para processos distintos de forma a mitigar vulnerabilidades internas do próprio ambiente de execução (Jiang e Hua 2025).

O interesse industrial por este modelo observa-se também em propostas recentes que procuram executar cargas JavaScript/Node.js dentro de *sandboxes* Wasm. Um exemplo é o Edge.js, apresentado pela Wasmer como um *runtime* compatível com Node.js que usa WebAssembly e WASIX para isolar chamadas de sistema e código nativo, preservando compatibilidade com aplicações existentes (Akbari 2026). Embora este caso se situe sobretudo em cenários de *edge*, Inteligência Artificial (IA) e *serverless*, reforça a tendência de usar Wasm como substrato prático de isolamento para cargas JavaScript fora do modelo tradicional de contentores (Wasmer, Edge.js, 2026).

3.3.3 Isolamento para Ambientes de Execução JavaScript

A segurança na execução de JavaScript tem evoluído através de diferentes estratégias de isolamento, tanto no lado do servidor como no lado do cliente. Esta distinção é importante, pois embora os ambientes difiram na sua arquitetura, partilham o objetivo crítico de confinar a execução de código não confiável e mitigar o impacto de dependências comprometidas.

Isolamento no Lado do Servidor

No contexto do lado do servidor, o Node.js estabeleceu-se operando sob um modelo de confiança implícita, onde o código executado herda a totalidade dos privilégios do utilizador do sistema operativo. Esta abordagem, focada na eficiência de I/O, delega a segurança para ferramentas externas, expondo a aplicação a riscos severos na cadeia de abastecimento.

Ambientes de execução de segunda geração, como o Deno, tentaram resolver esta lacuna implementando um modelo de controlo de acesso obrigatório, onde o acesso a recursos sensíveis requer concessão explícita via *flags*. Contudo, investigações recentes expuseram que este modelo de permissões sofre de uma granularidade insuficiente, já que uma permissão de leitura concedida a uma aplicação aplica-se indiscriminadamente a todas as suas dependências, permitindo que uma biblioteca maliciosa explore essa permissão partilhada para exfiltrar dados (AlHamdan e Staicu 2025).

Para responder a estas limitações de granularidade, surgiram soluções como o Cage4Deno. Esta ferramenta aborda especificamente o risco da execução de subprocessos, introduzindo uma camada de virtualização leve baseada em primitivas do Linux. O Cage4Deno permite definir políticas de segurança distintas para cada subprocesso invocado, criando *sandboxes* de maior granularidade que impedem a escalada de privilégios (Abbadini et al. 2023). Embora eficaz, esta abordagem depende fortemente de funcionalidades específicas do kernel Linux, limitando a sua portabilidade e aplicabilidade direta em ambientes de browser.

A recorrência de vulnerabilidades em *sandboxes* JavaScript reforça a fragilidade deste tipo de fronteira quando usada como mecanismo forte contra código malicioso. Por exemplo, a biblioteca vm2, usada para executar JavaScript não confiável em Node.js, voltou a ter vulnerabilidades críticas de escape de *sandbox* reportadas em 2026, incluindo CVE-2026-22709 (National Vulnerability Database 2026). Este tipo de incidente não invalida o valor de *sandboxes* JavaScript para certos cenários, mas ilustra a dificuldade de construir isolamento robusto apenas por interposição dentro do próprio ecossistema JavaScript.

Isolamento no Lado do Cliente

No domínio do *front-end* o desafio de isolar scripts de terceiros sem afetar a funcionalidade da página é complexo. Agten et al. (2012) apresentaram uma solução designada JSand, que implementa isolamento completo no lado do cliente. O JSand isola scripts não confiáveis executando-os dentro de um intérprete virtualizado escrito em JavaScript, interceptando e validando todas as interações com o Document Object Model (DOM) e APIs do browser (Agten et al. 2012). A implementação desta solução alicerça o isolamento através de um sistema de *object-capabilities* baseado num subconjunto de JavaScript designado Secure ECMAScript, que restringe a linguagem a um modelo de capacidades de objeto seguro. O isolamento é concretizado através do padrão Membrana, que é implementado através da colocação de *wrappers* em volta de objetos que providenciam operações sensíveis (Agten et al. 2012).

Desta forma a arquitetura assegura a mediação completa de todas as interações entre o script isolado e o ambiente do browser, operando sob um modelo de Allow-Listing. Neste modelo, o script recebe apenas um conjunto mínimo de referências infalsificáveis necessárias para a sua execução, sendo as restantes capacidades negadas por omissão (Agten et al. 2012). Isto permite garantir isolamento sem exigir modificações no browser do utilizador, provando ser tecnicamente viável conter código malicioso no cliente.

No entanto, o modelo do JSand evidencia o desempenho como o principal obstáculo. A virtualização de um intérprete JavaScript dentro de outro acarreta uma sobrecarga computacional significativa, tornando lenta a execução de aplicações complexas (Agten et al. 2012). De acordo com as medições realizadas pelos autores, o tempo de carregamento da biblioteca jQuery passa de 53.0ms num ambiente normal para 1458.2ms, um incremento de 1350.6ms. Já no caso de chamadas que cruzem a membrana, o tempo de execução pode ter

um incremento de 9 vezes quando comparando com um ambiente fora da *sandbox* (Agten et al. 2012). É neste ponto que o Wasm se distingue como uma alternativa superior. Ao contrário do JSand, que depende de interpretação lenta, o Wasm oferece isolamento ao nível da instrução binária com desempenho próximo do nativo, permitindo a criação de *sandboxes* seguras no browser sem a penalização de desempenho associada às técnicas de virtualização interpretada (Haas et al. 2017).

ShadowRealm

A proposta *ShadowRealm*, atualmente no estágio 2.7 do processo TC39, introduz uma primitiva ECMAScript para criar um novo *realm*: um ambiente com objeto global, objetos intrínsecos, cadeias de protótipos e grafo de módulos próprios (TC39 2025, 2026a). O método `evaluate()` permite avaliar código de forma síncrona nesse ambiente, enquanto `importValue()` carrega um módulo de forma assíncrona e devolve uma *Promise* resolvida com um dos seus *bindings*. A fronteira restringe os resultados observáveis a valores primitivos ou funções mediadas por *wrappers*, e a tentativa de transferir diretamente objetos não chamáveis produz um erro de tipo (TC39 2025, 2026b).

Este mecanismo oferece separação lógica dos objetos globais e intrínsecos, mas não define uma *heap* independente, uma política de capacidades ou limites de recursos. O código executa na mesma *thread*, podendo uma computação não terminante ou excessivamente pesada bloquear a *thread* e impedir o progresso da aplicação. A autoridade efetiva do código continua, porém, a depender das APIs adicionadas pelo ambiente hospedeiro ao objeto global e dos módulos que esse ambiente permite carregar. (TC39 2026b). Conceptualmente, um *ShadowRealm* aproxima-se mais de um *Context* V8 do que de um *Isolate*: ambos separam ambientes globais dentro de uma instância do motor, ao passo que cada *Isolate* mantém estado e *heap* gerido próprios e pode conter vários *Contexts* (V8 Project 2026a,b).

Face ao *ShadowRealm*, o Wasm estabelece uma fronteira estrutural mais forte onde o módulo tem acesso apenas às capacidades explicitamente importadas, e os acessos à memória linear são sujeitos a verificações de limites que impedem o acesso ao restante espaço de endereçamento do processo hospedeiro (Haas et al. 2017; WebAssembly Community Group 2025, 2026). Assim, embora o *ShadowRealm* seja útil para separar estados globais e reduzir a partilha acidental de objetos JavaScript, o Wasm permite controlar com maior granularidade a interface exposta e oferece isolamento de memória mais robusto para a execução de código não confiável.

3.4 Comparação e Posicionamento da Solução

Esta secção encerra o capítulo através de uma análise comparativa direta entre o isolamento baseado em Wasm e soluções de isolamento proeminentes no ecossistema atual. O objetivo é evidenciar lacunas que permanecem em aberto e avaliar como o Wasm se posiciona face a alternativas existentes. A subsecção 3.4.1 compara o WebAssembly com os V8 Isolates, discutindo os compromissos entre a facilidade de integração e o custo de memória por instância. De seguida, a subsecção 3.4.2 examina o Cage4Deno, uma solução académica recente para o ambiente Deno, contrastando a sua abordagem de isolamento baseada em processos e funcionalidades do *kernel* Linux com o modelo de isolamento do Wasm. Por fim, a subsecção 3.4.3 analisa o JSand, uma ferramenta de isolamento para JavaScript no lado do cliente, para demonstrar como o WebAssembly pode oferecer garantias de segurança equivalentes ou superiores, mas com menor sobrecusto de execução.

3.4.1 WebAssembly vs V8 Isolates

A arquitetura de V8 Isolates, amplamente adotada por plataformas de Edge Computing como a Cloudflare Workers, representa uma evolução significativa no modelo de isolamento multitenente, privilegiando a eficiência de recursos em detrimento da separação física tradicional. Tecnicamente, um Isolate é uma instância lógica independente do motor V8 que opera dentro de um único processo do sistema operativo anfitrião (Cloudflare 2025). Ao contrário de máquinas virtuais ou contentores, que duplicam todo o ambiente de execução, os Isolates alojados no mesmo processo partilham a implementação e parte da infraestrutura e do código do motor. Contudo, cada Isolate mantém estado e *heap* gerido próprios, sobre os quais opera a recolha dos respetivos objetos (Gruber 2018; V8 Project 2026a,b).

Do ponto de vista arquitetural, cada Isolate mantém o seu próprio *heap* de memória e um ou mais Contextos de execução (Google 2026c). Um Contexto no V8 corresponde a um ambiente de variáveis globais distinto, permitindo que múltiplas aplicações coexistam dentro do mesmo Isolate sem interferência direta no espaço de nomes (Google 2026c). Esta granularidade permite tempos de inicialização na ordem dos milissegundos e uma redução drástica no sobrecusto de memória, visto que não há necessidade de trocas de contexto ao nível do sistema operativo para alternar entre inquilinos (Cloudflare 2025).

Apesar da eficiência, este modelo de isolamento lógico apresenta desafios de segurança distintos. Embora os Isolates garantam a separação dos *heaps* de memória — mitigando os riscos de poluição de protótipos comuns em ambientes Node.js tradicionais —, a ausência de barreiras físicas ao nível do processo mantém o sistema dependente da correção do motor V8. Visto que múltiplos Isolates partilham o mesmo espaço de endereçamento virtual do processo hospedeiro, qualquer vulnerabilidade crítica no motor pode permitir que um inquilino ultrapasse as fronteiras lógicas do seu contexto e comprometa a integridade dos restantes (Cloudflare 2025; Google 2026c). Adicionalmente, esta partilha de recursos microarquiteturais expõe o sistema a ataques de canal lateral, como o Spectre, cuja mitigação eficaz em processos únicos exige técnicas complexas de scheduling e isolamento temporal que podem afetar o desempenho (Cloudflare 2025).

O WebAssembly distingue-se neste cenário ao oferecer um mecanismo de isolamento estruturalmente mais robusto, mesmo quando executado dentro do mesmo processo hospedeiro. A diferença fundamental reside no modelo de acesso à memória: enquanto os V8 Isolates dependem da lógica complexa do compilador JIT para gerir referências a objetos dispersos no *heap*, o que dificulta a verificação formal de segurança, o WebAssembly confina a execução a uma Memória Linear (Haas et al. 2017).

Esta memória é instanciada como um array de bytes contíguo, isolado do restante espaço de endereçamento do processo. O acesso aos dados é realizado exclusivamente através de índices numéricos relativos ao início deste array, permitindo que o motor de execução aplique verificações de limites determinísticas e eficientes. Em muitas implementações, esta verificação é reforçada pelo próprio *hardware* através de técnicas de memória virtual, que provocam uma exceção ao nível da Central Processing Unit (CPU) caso o módulo tente aceder a memória fora da sua *sandbox*. Desta forma, o Wasm reduz drasticamente a superfície de ataque para vulnerabilidades de corrupção de memória, oferecendo garantias de isolamento intra-processo superiores às do JavaScript puro (Haas et al. 2017).

3.4.2 WebAssembly vs Cage4Deno

Embora o foco deste trabalho seja o isolamento de dependências no contexto do browser, a inclusão do Cage4Deno nesta análise comparativa considera-se relevante. O Cage4Deno constitui um exemplo de reforço de segurança num ambiente de execução moderno — o Deno — que, apesar de ter sido desenhado com a segurança como pilar fundamental, enfrenta desafios arquiteturais ao interagir com componentes externos. A sua análise é pertinente pois ilustra uma abordagem de defesa em profundidade focada no controlo de elementos que operam fora da *sandbox* padrão. A comparação com este sistema permite, assim, contrastar uma abordagem de segurança baseada na restrição de processos ao nível do sistema operativo com o modelo de isolamento intra-processo e de gestão de memória oferecido pelo WebAssembly, contribuindo para uma melhor compreensão das diferentes camadas de proteção disponíveis para as arquiteturas de software modernas.

O Cage4Deno (Abbadini et al. 2023) representa desta forma uma proposta relevante para mitigar uma das principais superfícies de ataque em ambientes de execução JavaScript baseados em V8: a execução de subprocessos. Enquanto o Deno impõe um modelo de permissões restritivo ao código JavaScript executado dentro do motor V8, essa granularidade perde-se quando a aplicação invoca binários externos através de APIs como `Deno.command`. Nesses cenários, o subprocesso herda, por defeito, as permissões do utilizador do sistema operativo, escapando à *sandbox* lógica do ambiente de execução. O Cage4Deno resolve esta lacuna usando ferramentas do *kernel* Linux para impor políticas de acesso a esses processos filho.

A diferença fundamental entre o Cage4Deno e o WebAssembly reside na unidade de isolamento e no mecanismo de confinamento. O Cage4Deno opera ao nível do processo do sistema operativo. Conforme ilustrado na Figura 3.3, para isolar uma tarefa, o sistema depende das primitivas de segurança do *kernel* para restringir o que o binário executável pode ver ou fazer (Abbadini et al. 2023). Embora eficaz em ambientes Linux, esta abordagem herda o peso da gestão de processos. Cada tarefa isolada exige a criação de um novo processo, com o consequente sobrecusto de memória e tempo de escalonamento.

Em contraste, o WebAssembly propõe um modelo de isolamento intra-processo. Múltiplos módulos ou componentes Wasm podem ser instanciados dentro do mesmo processo hospedeiro, separados logicamente pelas suas memórias lineares independentes (Haas et al. 2017). Como cada módulo só consegue aceder ao seu próprio *array* de bytes contíguo, o isolamento é garantido matematicamente pelo ambiente de execução, sem necessidade de recorrer a trocas de contexto do sistema operativo. Isto permite uma densidade de execução superior à abordagem baseada em processos do Cage4Deno.

Um ponto importante é a utilização do motor V8. Tanto o Deno, que serve de base ao Cage4Deno, como a outros ambientes de execução de servidor utilizam o motor V8 (Project 2026) para executar código JavaScript. No modelo Deno/Cage4Deno, o V8 é usado para executar a lógica de orquestração, mas a execução de tarefas pesadas ou inseguras é frequentemente delegada a subprocessos nativos para evitar bloquear o ciclo de eventos (*event loop*) ou para usar ferramentas existentes (Abbadini et al. 2023). O Cage4Deno foca-se em proteger esses subprocessos.

O WebAssembly posiciona-se de forma diferente. Embora possa ser executado dentro do V8, o Wasm tenta evitar a necessidade de subprocessos nativos. Ao compilar bibliotecas para Wasm, estas podem correr dentro do mesmo processo, sujeitas às mesmas garantias de isolamento. Ou seja, enquanto o Cage4Deno tenta conter processos nativos inseguros, o

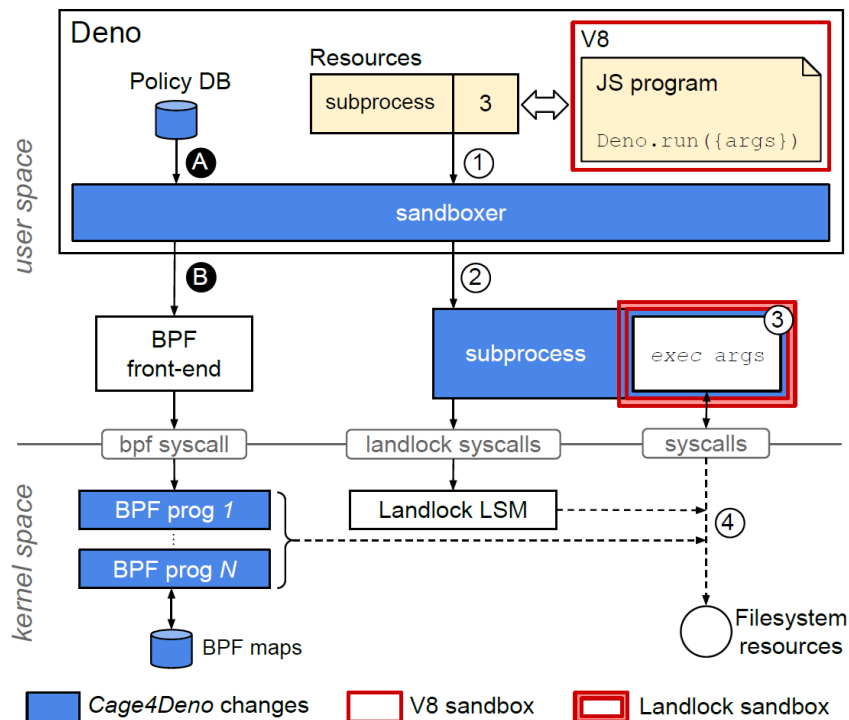


Figura 3.3: Arquitetura do Cage4Deno (Abbadini et al. 2023)

WebAssembly tenta trazer a computação para dentro de um ambiente gerido e seguro por concepção.

Existem também diferenças no modelo de permissões. O modelo de permissões do Cage4Deno é declarativo e imposto externamente: define-se uma política que o *kernel* aplica ao processo. Esta dependência do Linux torna a solução pouco portátil para outros sistemas operativos.

O WebAssembly, através do Component Model, adota um modelo de capacidades. Ao invés de pedir acesso a um ficheiro, o código Wasm recebe uma permissão para um ficheiro específico no momento da instanciação. O acesso a recursos é injetado explicitamente, não herdado implicitamente. Isto torna o modelo de segurança do Wasm agnóstico ao sistema operativo subjacente, funcionando de forma idêntica em browsers, servidores ou dispositivos embebidos, garantindo uma portabilidade que soluções baseadas em primitivas de sistema operativo, como o Cage4Deno, não conseguem oferecer.

Em suma, o Cage4Deno é uma solução robusta para o problema específico de proteger binários legados em ambientes Linux. Contudo, o WebAssembly oferece uma grande mudança arquitetural: move o isolamento do nível do sistema operativo para o nível da aplicação. Ao transformar dependências binárias em módulos Wasm confinados, elimina-se a classe de vulnerabilidades que o Cage4Deno visa mitigar, oferecendo uma solução de segurança mais integrada e eficiente para arquiteturas de *software* modernas.

3.4.3 WebAssembly vs JSand

A evolução das técnicas de isolamento no lado do cliente atingiu um marco importante com a proposta do JSand por Agten et al. (2012), que demonstrou a viabilidade de confinar código JavaScript de terceiros utilizando apenas primitivas da linguagem sem modificações no

browser. Embora o JSand represente um marco importante em isolamento puramente aplicativo, a sua eficácia está intrinsecamente limitada pela natureza dinâmica e permissiva da linguagem JavaScript. A arquitetura do WebAssembly introduz uma mudança de paradigma, transitando de uma segurança baseada em interposição para um modelo de segurança por concepção (*security-by-design*).

A distinção mais crítica reside na gestão de memória. O JSand opera num *heap* partilhado, obrigando à interposição de todas as interações entre o código confinado e o ambiente global (Agten et al. 2012). Esta abordagem, embora funcional, é frágil. Uma única falha na lógica dos *wrappers* pode comprometer todo o isolamento. Em contraste, o WebAssembly impõe um isolamento estrutural através da memória linear. Como notado por Haas et al. (2017) e reforçado por Kim, Jang e Shin (2022) na sua análise de segurança, o acesso à memória num módulo Wasm é confinado a um *array* de bytes contíguo, desprovido de acesso direto aos objetos do hospedeiro.

Mais do que o isolamento de dados, o WebAssembly oferece garantias estruturais que dificultam a execução de ataques clássicos de corrupção de memória. A arquitetura separa o código executável dos dados, impedindo que dados injetados na memória linear sejam executados como instruções (Haas et al. 2017; Kim, Jang e Shin 2022). Adicionalmente, a pilha de execução (*call stack*) é gerida pelo motor fora do espaço de endereçamento acessível pelo programa, o que torna inviáveis os ataques tradicionais de *stack smashing* que tentam sobrescrever endereços de retorno, uma vulnerabilidade que em ambientes nativos é de mitigação complexa.

Para além do isolamento de memória, como se pode observar na Figura 3.4, o JSand define políticas de acesso para filtrar o que um *script* pode ou não fazer usando uma abordagem de lista de negação/permissão sobre APIs globais. Por outro lado, o ecossistema WebAssembly evoluiu para um modelo de segurança baseado em capacidades, especialmente com a introdução do Modelo de Componentes.

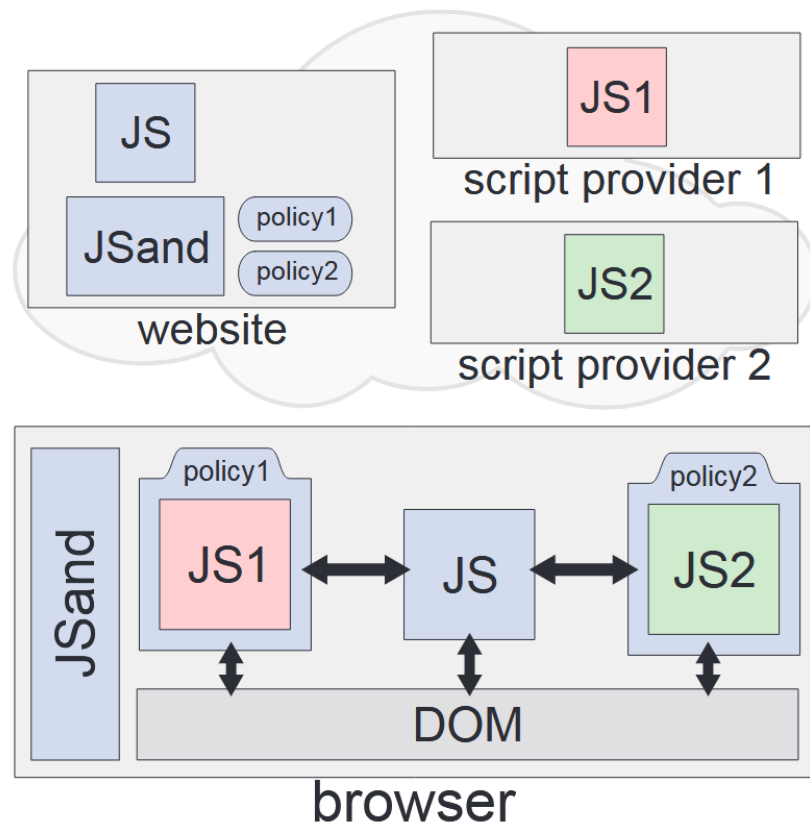


Figura 3.4: Arquitetura do JSand (Agten et al. 2012)

Neste modelo, um módulo Wasm não herda implicitamente o acesso a APIs globais ou ao sistema operativo. Em vez disso, o acesso a recursos - desde I/O à rede - deve ser explicitamente injetado no módulo pelo hospedeiro no momento da instanciação (Alliance 2025d,e). Isto permite uma granularidade de permissões superior à do JSand. Por exemplo, é possível fornecer a um módulo uma capacidade para escrever apenas num diretório específico, sem que o módulo tenha sequer conhecimento da existência de outras partes do sistema de ficheiros. Esta abordagem de "princípio do menor privilégio" é estrutural e verificável estaticamente através das interfaces WIT dos componentes, contrastando com a verificação dinâmica e propensa a erros das políticas do JSand.

Em suma, embora o JSand tenha provado que é possível isolar JavaScript sem alterações ao browser, o WebAssembly oferece uma fundação superior para a execução de código não confiável. A combinação de isolamento de memória robusto via memória linear, proteção da pilha de execução e um modelo de capacidades granular posiciona o WebAssembly não apenas como uma alternativa com desempenho superior, mas como um mecanismo de segurança mais resiliente para arquiteturas de *software* modernas.

A Tabela 3.2 resume os atributos principais das técnicas de isolamento discutidas nesta secção.

Tabela 3.2: Comparação entre técnicas de isolamento

Atributo	WebAssembly	V8 Isolates	Cage4Deno	JSand
Memória	Linear isolada	Gestão de referências	Processo isolado	Partilhada
Sobrecusto	Próximo ao nativo	Baixo	Médio	Alto
Controlo de acesso	Capacidades	Contexto	Políticas de SO	Membrana
Granularidade	Módulo	Isolate intra-processo	Processo	Biblioteca/objeto

Capítulo 4

Análise

Este capítulo define os requisitos da solução proposta, partindo do problema identificado nos capítulos anteriores e dos objetivos formulados na Secção 1.3. A análise centra-se nas capacidades que a solução deve disponibilizar, nos atributos de qualidade que deve satisfazer e nas interações principais que devem ser suportadas.

No contexto desta dissertação, a solução deve responder à ausência de isolamento granular para bibliotecas JavaScript/npm executadas no contexto privilegiado de aplicações Web. Uma dependência comprometida pode herdar permissões do ambiente hospedeiro, aceder a capacidades sensíveis e afetar o comportamento da aplicação como um todo. Assim, os requisitos definidos neste capítulo procuram expressar o compromisso central da solução: reforçar a segurança através de isolamento e controlo explícito de capacidades, preservando níveis aceitáveis de compatibilidade, desempenho e usabilidade.

O capítulo organiza-se em três partes. Primeiro, definem-se os requisitos funcionais, que descrevem as capacidades essenciais da solução. Depois, apresentam-se os requisitos não funcionais, organizados segundo o modelo FURPS+. Por fim, descrevem-se os casos de uso principais, que sintetizam as interações fundamentais entre o desenvolvedor, a aplicação e o mecanismo de isolamento.

4.1 Requisitos Funcionais

Os requisitos funcionais descrevem as capacidades que a solução deve disponibilizar para concretizar o isolamento granular de dependências. Dado que o foco do trabalho é a segurança em tempo de execução, estes requisitos descrevem as capacidades nucleares que o mecanismo de isolamento deve fornecer para que uma aplicação consiga usar bibliotecas JavaScript de forma confinada e controlada.

A Tabela 4.1 apresenta os requisitos funcionais considerados essenciais.

Tabela 4.1: Requisitos funcionais da solução

ID	Requisito
RF-01	A solução deve permitir selecionar quais bibliotecas JavaScript são executadas em isolamento.
RF-02	A solução deve permitir que a aplicação utilize bibliotecas isoladas mantendo, tanto quanto possível, a forma habitual de importação e uso.
RF-03	A solução deve confinar a execução da biblioteca isolada a um ambiente separado do contexto principal da aplicação.
RF-04	A solução deve permitir definir, por biblioteca, quais capacidades do ambiente hospedeiro podem ser acedidas.
RF-05	A solução deve bloquear o acesso a capacidades não autorizadas.

O requisito RF-01 estabelece a granularidade pretendida: a unidade de isolamento é a biblioteca, e não a aplicação completa. Esta decisão decorre diretamente do problema em estudo, uma vez que o risco introduzido pela cadeia de abastecimento se manifesta através de dependências específicas que podem ser legítimas, vulneráveis ou comprometidas.

O RF-02 introduz uma preocupação de compatibilidade e adoção. Uma solução que exigisse reescrita extensa da aplicação reduziria a sua utilidade prática, mesmo que oferecesse isolamento forte. Assim, a aplicação deve continuar a interagir com as bibliotecas isoladas de uma forma próxima do uso normal de dependências JavaScript.

Os requisitos RF-03, RF-04 e RF-05 definem o núcleo de segurança da solução. A biblioteca isolada deve executar fora do contexto principal da aplicação, deve receber apenas as capacidades explicitamente configuradas e deve ser impedida de aceder a capacidades não autorizadas. Em conjunto, estes requisitos materializam uma abordagem baseada no princípio do menor privilégio, onde a biblioteca não deve herdar automaticamente todas as permissões do ambiente onde a aplicação executa.

4.2 Requisitos Não Funcionais

Os requisitos não funcionais descrevem atributos de qualidade e restrições que influenciam a forma como a solução deve ser concebida. Neste trabalho, estes requisitos são organizados segundo o modelo FURPS+, que agrupa preocupações de funcionalidade, usabilidade, fiabilidade, desempenho, suportabilidade e restrições adicionais (Kruchten 2004).

Neste projeto, a segurança é a dimensão dominante, por estar diretamente ligada ao objetivo de mitigar o impacto de dependências não confiáveis. A usabilidade e o desempenho são também relevantes, pois condicionam a adoção prática da solução. A suportabilidade é considerada na medida em que a abordagem deve poder evoluir para diferentes bibliotecas e padrões de utilização. Por fim, a categoria “+” é usada para explicitar restrições de contexto e de âmbito.

4.2.1 Functionality / Segurança

A categoria de funcionalidade, no contexto do modelo FURPS+, inclui capacidades e propriedades observáveis do sistema. Neste trabalho, a principal propriedade funcional de qualidade é a segurança. A solução deve reduzir a autoridade efetiva das bibliotecas isoladas, impedir

efeitos diretos sobre o contexto hospedeiro e garantir que as permissões são concedidas apenas quando necessárias.

Tabela 4.2: Requisitos não funcionais de segurança

ID	Requisito
RNF-S01	A solução deve reduzir a superfície de ataque associada a dependências npm não confiáveis.
RNF-S02	A solução deve impedir que alterações de estado global dentro da biblioteca isolada afetem diretamente o ambiente hospedeiro.
RNF-S03	A política de permissões deve seguir o princípio do menor privilégio.

O RNF-S01 define o objetivo de segurança em termos de redução de superfície de ataque, e não de eliminação absoluta de risco. Esta formulação é importante porque uma solução de isolamento não substitui outras práticas de segurança, como auditoria de dependências ou atualização de pacotes. O seu contributo esperado é limitar o impacto de código não confiável durante a execução.

O RNF-S02 é particularmente relevante para ataques que exploram estado global partilhado. Se uma biblioteca isolada modificar o seu próprio ambiente de execução, essa alteração não deve propagar-se diretamente para a aplicação hospedeira. O RNF-S03 complementa esta exigência ao estabelecer que capacidades sensíveis devem ser negadas por omissão e concedidas apenas quando explicitamente necessárias.

4.2.2 Usability / Usabilidade

A usabilidade é considerada do ponto de vista do desenvolvedor que integra a solução numa aplicação Web. Como a solução atua sobre bibliotecas de terceiros, o custo de adoção deve ser baixo. Caso contrário, a segurança adicional tenderia a ser rejeitada em projetos reais por exigir alterações excessivas ao código existente ou por tornar a configuração difícil de compreender.

Tabela 4.3: Requisitos não funcionais de usabilidade

ID	Requisito
RNF-U01	A solução deve exigir o mínimo possível de alterações no código da aplicação.
RNF-U02	A configuração de isolamento e permissões deve ser explícita e compreensível para o desenvolvedor.

O RNF-U01 reforça que a adoção da solução deve preservar a experiência habitual de desenvolvimento sempre que possível. O desenvolvedor deve conseguir isolar bibliotecas selecionadas sem reestruturar a aplicação em torno do mecanismo de segurança. O RNF-U02 exige que a configuração seja explícita, pois decisões de segurança implícitas ou difíceis de auditar reduzem a confiança no comportamento da solução.

4.2.3 Performance / Desempenho

A introdução de uma fronteira de isolamento implica necessariamente custos adicionais. Chamadas entre a aplicação e a biblioteca isolada, transferência de dados, inicialização do ambiente e mediação de permissões podem aumentar o tempo de execução ou o consumo de recursos. Por isso, o requisito de desempenho não exige equivalência absoluta com a execução nativa, mas sim um sobrecusto aceitável para cenários de utilização realistas.

Tabela 4.4: Requisitos não funcionais de desempenho

ID	Requisito
RNF-P01	A solução deve introduzir sobrecusto aceitável face à execução nativa para cenários de uso realistas.

Este requisito é formulado de forma qualitativa nesta fase da análise. A definição operacional de sobrecusto, as métricas usadas e os cenários experimentais pertencem ao Capítulo 7. Neste capítulo, importa apenas estabelecer que a solução deve procurar um equilíbrio entre reforço de segurança e impacto prático no desempenho.

4.2.4 Supportability / Manutenção e Evolução

A suportabilidade diz respeito à capacidade de manter e evoluir a solução. Como o ecossistema npm é heterogêneo, a solução deve admitir extensão gradual a novas bibliotecas, novos tipos de dados e novos padrões de interação entre a aplicação e a dependência isolada.

Tabela 4.5: Requisitos não funcionais de manutenção e evolução

ID	Requisito
RNF-M01	A solução deve ser extensível a diferentes bibliotecas.
RNF-M02	A solução deve disponibilizar um mecanismo extensível de marshalling para preservar a semântica de tipos não suportados adequadamente pelo mecanismo de conversão genérico.

O RNF-M01 não implica suporte universal a todas as bibliotecas npm. Esse objetivo seria demasiado amplo para o âmbito desta dissertação. O requisito estabelece antes que a solução deve ser concebida de forma extensível, permitindo validar a abordagem com bibliotecas reais e evoluir posteriormente para cobrir casos adicionais.

O RNF-M02 decorre da heterogeneidade dos valores usados pelas bibliotecas JavaScript. Um mecanismo genérico de conversão não consegue garantir, por omissão, a preservação da representação e da semântica de todos os tipos que podem atravessar a fronteira de isolamento. Exemplos como `image-js` e `tweetnacl` recorrem a dados binários e `TypedArray`; `lodash` distingue objetos `Arguments` de arrays e objetos comuns; e bibliotecas de validação, como `zod` e `yup`, utilizam erros tipados.

Estes exemplos não definem uma lista exaustiva de tipos a suportar. Demonstram antes que a necessidade de tratamentos específicos é previsível e que a solução deve permitir acrescentá-los sem alterar a lógica principal de isolamento e comunicação. O requisito

estabelece, assim, um ponto de extensão para aumentar progressivamente a compatibilidade com novos tipos e bibliotecas.

4.2.5 Restrições

No modelo FURPS+, a categoria “+” inclui restrições de concepção, implementação, interface, operação ou contexto. Estas restrições não descrevem funcionalidades adicionais, mas delimitam o espaço em que a solução deve ser concebida. Para este trabalho, as restrições relevantes são apresentadas na Tabela 4.6.

Tabela 4.6: Restrições da solução

ID	Restrição
RST-01	A solução deve ser orientada ao contexto de aplicações JavaScript executadas no browser.
RST-02	A solução deve explorar Wasm como base para isolamento.
RST-03	A solução deve operar ao nível de bibliotecas/dependências, não ao nível de aplicações inteiras.

As restrições RST-01 e RST-03 delimitam o âmbito do trabalho. A solução é orientada ao contexto de aplicações Web e ao isolamento de bibliotecas ou dependências específicas, não ao isolamento de aplicações completas. A restrição RST-02 decorre da questão de investigação e posiciona o Wasm como a base tecnológica a explorar para isolamento. A forma concreta como esta restrição é satisfeita é discutida no Capítulo 5.

4.3 Casos de Uso

Os casos de uso representam as interações principais entre os atores envolvidos e a solução proposta. Dado o âmbito focado do projeto, são considerados apenas os casos de uso essenciais para expressar o objetivo central da solução: permitir isolamento granular de bibliotecas com controlo explícito de capacidades.

O ator principal é o desenvolvedor da aplicação, responsável por escolher que bibliotecas devem ser isoladas e que capacidades lhes devem ser atribuídas. A aplicação Web e a biblioteca JavaScript isolada participam nos casos de uso como entidades do sistema, mas não são tratadas como atores humanos. A Tabela 4.7 resume os casos de uso identificados.

Tabela 4.7: Casos de uso principais

ID	Caso de uso	Ator principal
UC-01	Configurar biblioteca para isolamento	desenvolvedor
UC-02	Usar biblioteca isolada na aplicação	desenvolvedor
UC-03	Conceder capacidade específica a uma biblioteca isolada	desenvolvedor

4.3.1 UC-01: Configurar Biblioteca para Isolamento

Este caso de uso representa a decisão inicial do desenvolvedor de executar uma biblioteca JavaScript em isolamento. O desenvolvedor identifica uma dependência que pretende confinar e declara essa intenção na configuração da solução. A partir dessa configuração, a solução deve tratar essa biblioteca como uma dependência não confiável e encaminhar a sua utilização através do mecanismo de isolamento.

A pré-condição principal é a existência de uma aplicação Web que utiliza pelo menos uma biblioteca JavaScript passível de ser isolada. O fluxo principal inicia-se quando o desenvolvedor seleciona a biblioteca, define a sua inclusão no conjunto de bibliotecas isoladas e executa o processo normal de preparação da aplicação. No final, a biblioteca fica marcada para execução isolada, sem que o desenvolvedor tenha de alterar manualmente todos os pontos onde essa biblioteca é usada.

Este caso de uso está diretamente relacionado com os requisitos RF-01, RF-02 e RNF-U01, pois combina a seleção granular de dependências com a necessidade de preservar a usabilidade da solução.

4.3.2 UC-02: Usar Biblioteca Isolada na Aplicação

Este caso de uso descreve a utilização normal de uma biblioteca depois de configurada para isolamento. O desenvolvedor escreve ou mantém código aplicativo que invoca a biblioteca de forma próxima da utilização habitual. A solução deve encaminhar essa utilização para o ambiente isolado e devolver os resultados esperados à aplicação.

A pré-condição é que a biblioteca tenha sido previamente configurada para isolamento. O fluxo principal ocorre quando a aplicação invoca uma função, classe ou objeto disponibilizado pela biblioteca. A solução deve mediar a chamada, executar a operação no ambiente isolado e disponibilizar o resultado ao código da aplicação. Caso a chamada envolva dados ou objetos que atravessam a fronteira de isolamento, a solução deve preservar a semântica necessária para a aplicação continuar funcional. Como fluxo alternativo, se a biblioteca tentar aceder a uma capacidade que não lhe foi concedida, a operação deve ser bloqueada de acordo com a política de isolamento.

Este caso de uso concretiza os requisitos RF-02 e RF-03. Também se relaciona com RNF-U01, RNF-M01 e RNF-M02, uma vez que a viabilidade da solução depende da sua capacidade de suportar padrões reais de utilização de bibliotecas e de preservar a semântica dos valores que atravessam a fronteira de isolamento.

4.3.3 UC-03: Conceder Capacidade Específica a uma Biblioteca Isolada

Este caso de uso representa a concessão explícita de uma capacidade do ambiente hospedeiro a uma biblioteca isolada. Algumas bibliotecas podem necessitar de acesso controlado a funcionalidades externas para executar corretamente. No entanto, esse acesso não deve ser herdado automaticamente pelo simples facto de a biblioteca estar integrada na aplicação.

O fluxo principal inicia-se quando o desenvolvedor identifica uma capacidade necessária e a declara na configuração da biblioteca. Durante a execução, quando a biblioteca tenta usar essa capacidade, a solução deve disponibilizá-la apenas se ela tiver sido autorizada. Capacidades não declaradas permanecem indisponíveis por omissão. Desta forma, o desenvolvedor mantém controlo explícito sobre a autoridade concedida a cada dependência.

Este caso de uso relaciona-se com RF-04, RNF-S01, RNF-S03 e RNF-U02. A sua importância reside no equilíbrio entre segurança e funcionalidade: a solução deve restringir permissões por omissão, mas permitir a concessão controlada de capacidades quando estas são necessárias.

4.4 Síntese da Análise

A análise definiu os requisitos funcionais de isolamento granular e de controlo explícito das capacidades concedidas às bibliotecas. Os requisitos não funcionais complementam essas capacidades com exigências de segurança, usabilidade, desempenho e extensibilidade, enquanto os casos de uso as traduzem nas principais interações do desenvolvedor com a solução. Este enquadramento estabelece as necessidades que orientam as decisões arquiteturais apresentadas no capítulo seguinte.

Capítulo 5

Design da Solução

Este capítulo apresenta a conceção da solução proposta para o isolamento granular de bibliotecas JavaScript/npm no browser. Partindo dos requisitos definidos no Capítulo 4, descreve as decisões arquiteturais que permitem integrar o isolamento no processo de construção, executar cada biblioteca num contexto QuickJS/Wasm dedicado e mediar explicitamente a comunicação com o ambiente hospedeiro.

Inicialmente, a Secção 5.1 apresenta as principais vistas arquiteturais e os componentes da solução. As Secções 5.2 e 5.3 descrevem, respetivamente, a transformação das dependências durante o processo de construção e a sua execução no ambiente isolado.

A Secção 5.4 detalha o controlo das capacidades concedidas e as membranas que medeiam a comunicação entre o ambiente hospedeiro e o ambiente isolado. De seguida, a Secção 5.5 apresenta extensões orientadas para o desempenho e para a evolução do marshalling. Por fim, a Secção 5.6 sintetiza as decisões arquiteturais e relaciona-as com os requisitos da solução.

A organização dos diagramas segue a abordagem hierárquica do modelo C4, na qual cada nível representa um aumento da granularidade da descrição arquitetural (Brown 2026).

5.1 Visão Geral da Arquitetura

A solução proposta introduz uma camada de isolamento entre a aplicação hospedeira e as bibliotecas JavaScript selecionadas pelo desenvolvedor. Em vez de alterar diretamente o código da aplicação ou exigir que o desenvolvedor invoque manualmente uma API de isolamento, a solução atua sobre o processo de construção da aplicação. As bibliotecas configuradas para isolamento são substituídas por *proxies* gerados, que preservam a interface esperada pela aplicação e encaminham as interações para um ambiente QuickJS executado sobre Wasm.

A Figura 5.1 apresenta a vista lógica de nível dois dos componentes da solução. O seu objetivo é identificar os componentes arquiteturais que pertencem à solução e mostrar como estes colaboram para cumprir os objetivos de isolamento, compatibilidade e controlo de capacidades.

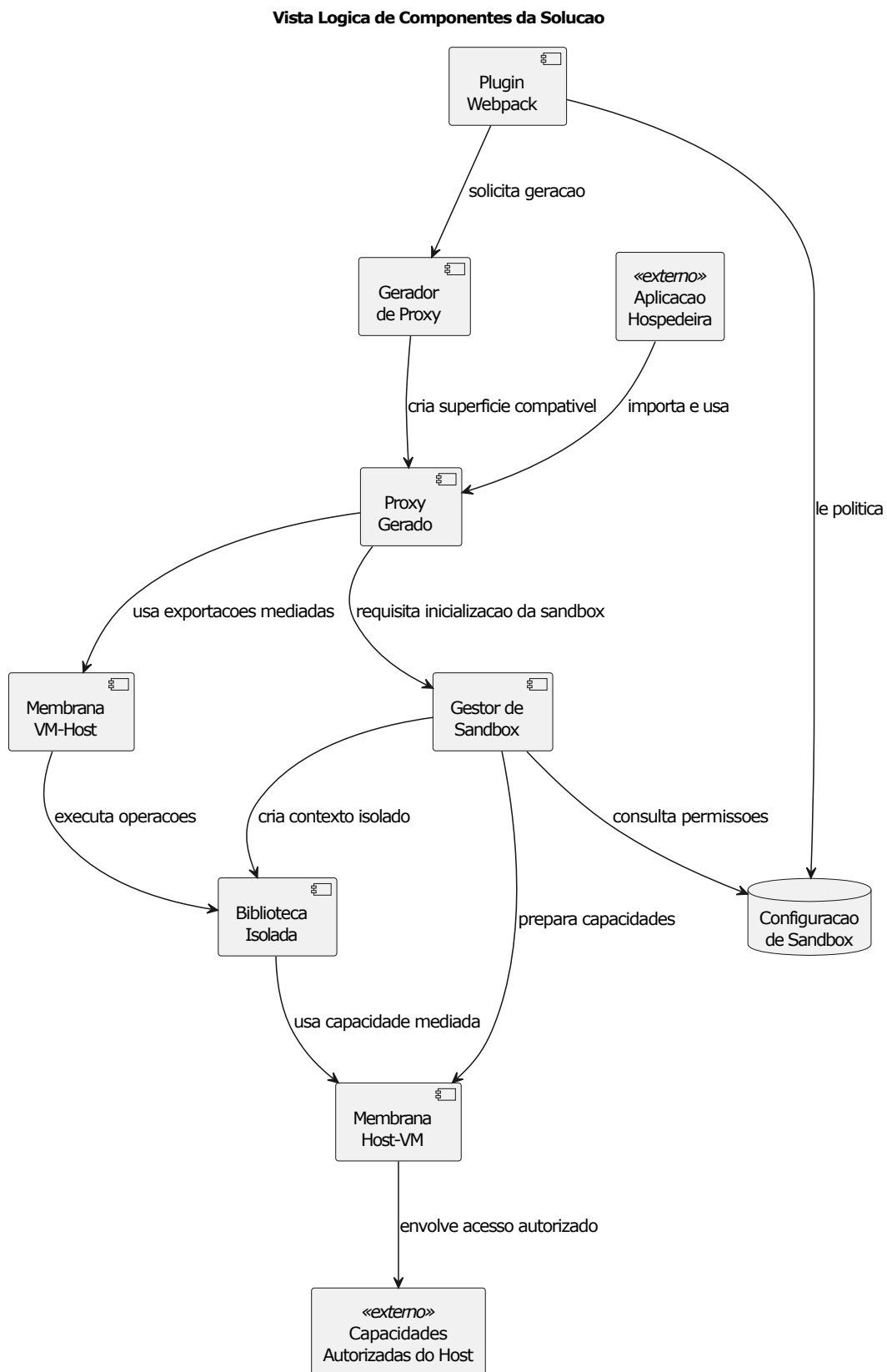


Figura 5.1: Vista lógica de componentes da solução

O componente *Plugin Webpack* representa o ponto de integração da solução com o processo de construção. Este componente consulta a Configuração da *Sandbox* para identificar que bibliotecas devem ser isoladas, aciona o Gerador de *Proxy* e estabelece o redirecionamento necessário para que a aplicação deixe de importar diretamente a implementação original da dependência.

A *Configuração da Sandbox* constitui o ponto declarativo da solução. É neste componente que o desenvolvedor indica as bibliotecas que devem executar de forma isolada e as capacidades do ambiente hospedeiro que cada uma pode utilizar. Esta separação é relevante porque mantém as decisões de segurança fora do código da aplicação, tornando explícito que o acesso a capacidades externas deve ser concedido de forma controlada e por biblioteca.

O *Gerador de Proxy* produz o componente que será importado pela aplicação no lugar da biblioteca original. Para isso, a implementação recorre a informação obtida a partir da interface pública da biblioteca, gerando a partir daí uma superfície compatível com a forma de utilização esperada pela aplicação.

O *Proxy Gerado* funciona como a fronteira direta para a aplicação hospedeira. Do ponto de vista da aplicação, este componente representa a biblioteca importada e disponibiliza uma interface compatível com a utilização esperada. Internamente, contudo, o *proxy* delega a inicialização para o *Gestor de Sandbox* e usa as referências mediadas devolvidas por esse gestor para encaminhar operações posteriores. Esta separação preserva a usabilidade da solução ao permitir que a aplicação continue a usar a biblioteca de forma próxima do uso normal, enquanto a execução real da dependência é deslocada para um ambiente isolado.

O *Gestor de Sandbox* é responsável por criar e coordenar o ambiente de execução isolado. Este componente consulta as permissões associadas à biblioteca, prepara as capacidades autorizadas do ambiente hospedeiro e cria o contexto QuickJS/Wasm onde a biblioteca será executada. Assim, o gestor centraliza a ligação entre a política declarada, o *proxy* usado pela aplicação e o ambiente isolado onde a biblioteca efetivamente é executada.

As *Membranas Host-VM* e *VM-Host* representam a fronteira bidirecional de comunicação. A primeira controla a exposição de capacidades autorizadas do ambiente hospedeiro à biblioteca isolada; a segunda preserva a utilização da API da biblioteca pela aplicação, mantendo funções e objetos reais no contexto isolado sempre que necessário. Estas membranas fazem parte da conceção base, porque o isolamento só é útil se a comunicação indispensável for mediada de forma explícita.

A *Biblioteca Isolada* corresponde à dependência npm cuja execução deixa de ocorrer diretamente no mesmo contexto JavaScript da aplicação hospedeira. Em vez disso, o código da biblioteca é avaliado dentro do contexto QuickJS/Wasm. Consequentemente, uma biblioteca isolada não deve conseguir aceder diretamente ao estado interno da aplicação, a outras bibliotecas isoladas, ou a capacidades do ambiente hospedeiro que não tenham sido explicitamente expostas. As *Capacidades Autorizadas do Host* representam precisamente o subconjunto de APIs ou objetos do ambiente hospedeiro que atravessa a fronteira depois de validado pela configuração e envolvido pela membrana apropriada.

A Figura 5.2 apresenta a vista de implementação de nível três. Esta vista complementa a vista lógica ao mostrar como as responsabilidades arquiteturais são organizadas no projeto. Ao contrário da vista lógica, o objetivo aqui é evidenciar a estrutura modular por pastas principais.

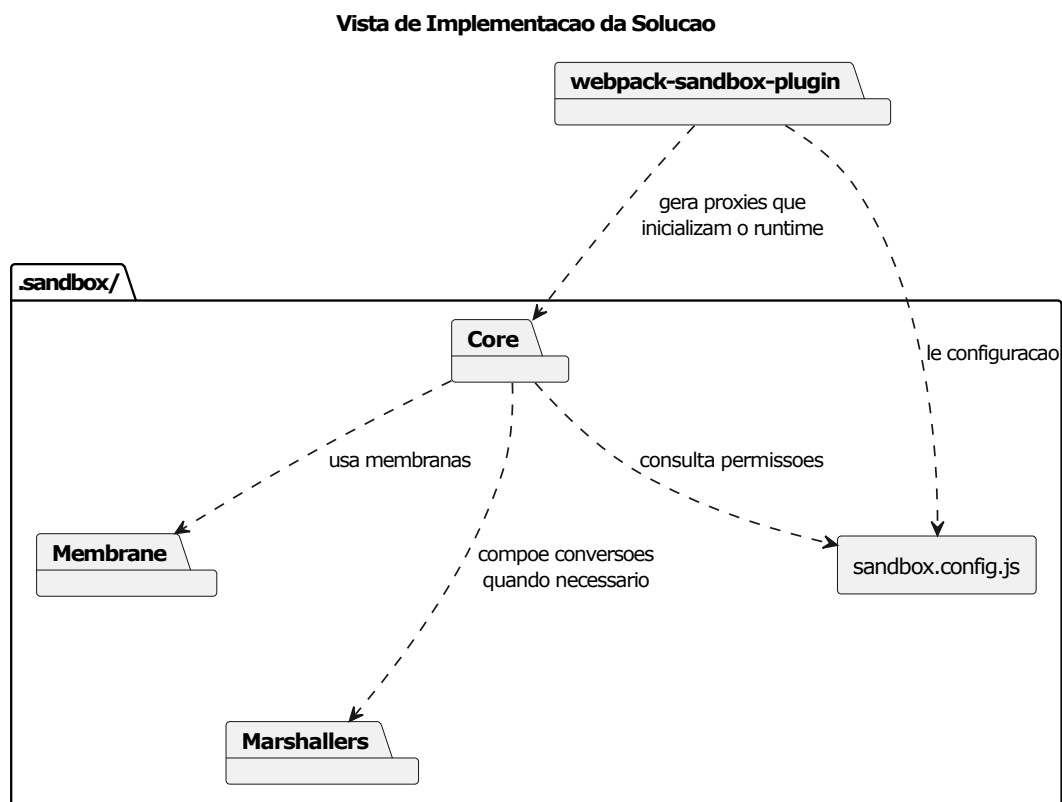


Figura 5.2: Vista de implementação da solução

Na implementação, **webpack-sandbox-plugin** concentra a integração com o processo de construção e a geração dos *proxies*. A pasta **.sandbox/core/** agrupa a lógica de criação e coordenação das *sandboxes* em tempo de execução. A pasta **.sandbox/membrane/** contém as duas direções de mediação da fronteira entre hospedeiro e VM. A pasta **.sandbox/marshallers/** contém extensões de conversão usadas quando determinados tipos não podem atravessar a fronteira pelo mecanismo predefinido. Por fim, **.sandbox/sandbox.config.js** materializa a política declarativa de isolamento e permissões.

A Figura 5.3 complementa as vistas anteriores com uma vista de implantação de nível dois. Esta vista facilita a distinção de que: o plugin e o gerador atuam no processo de construção, o *proxy*, o gestor e as membranas executam no motor JavaScript do ambiente hospedeiro, e a biblioteca isolada passa a executar no contexto QuickJS/Wasm.

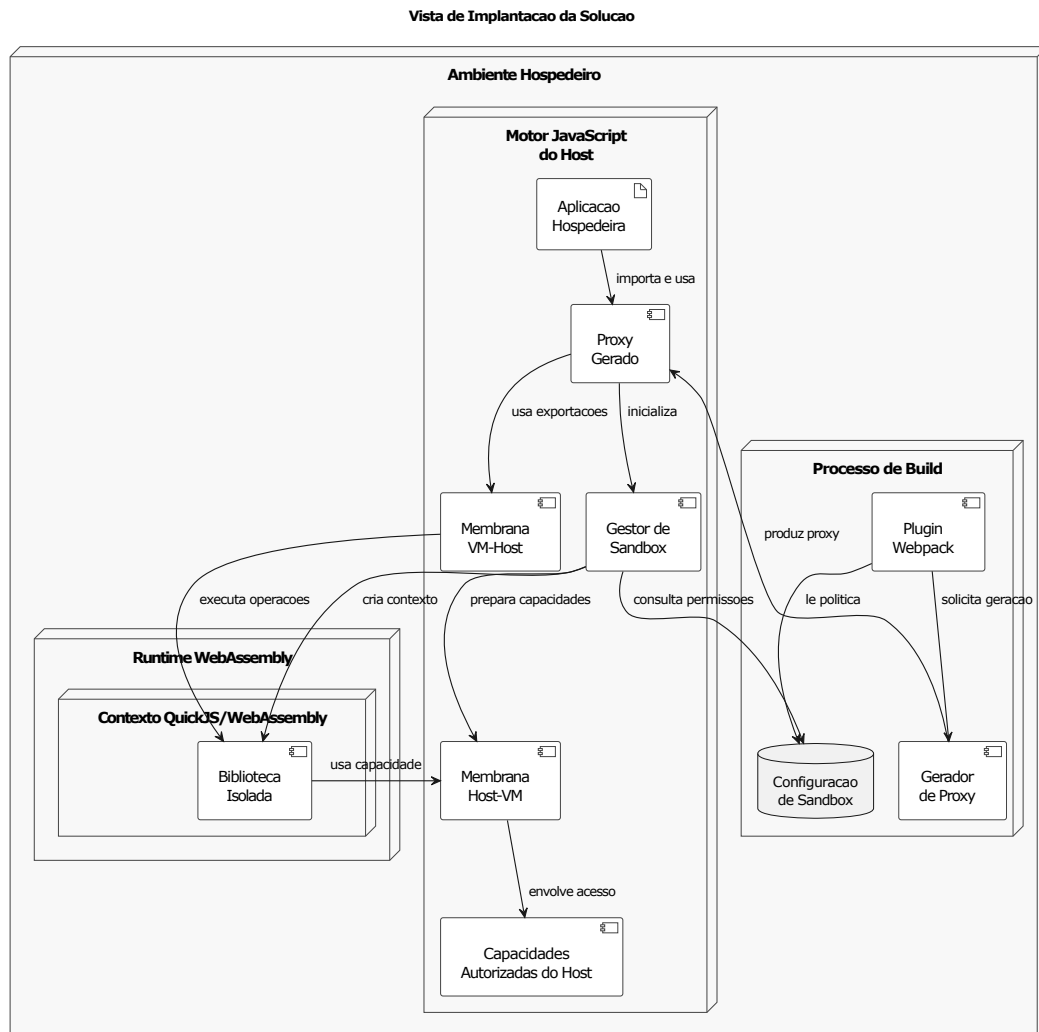


Figura 5.3: Vista de implantação da solução

Nesta vista, a aplicação hospedeira e o *Proxy Gerado* permanecem no ambiente JavaScript do ambiente hospedeiro. O *proxy* comunica com a *sandbox* através das membranas e das referências mediadas devolvidas durante a inicialização. A execução da biblioteca configurada ocorre no contexto QuickJS alojado em Wasm, e as capacidades do ambiente hospedeiro só atravessam a fronteira quando são explicitamente autorizadas pela Configuração da *Sandbox*.

Em síntese, a arquitetura separa a solução em dois momentos complementares. No momento de construção (*build*), o plugin integra-se no processo de construção da aplicação e substitui as dependências configuradas por *proxies*. Em tempo de execução, os *proxies* recorrem ao Gestor de *Sandbox* para executar a biblioteca dentro de um contexto QuickJS/Wasm e para mediar o acesso ao ambiente hospedeiro. As secções seguintes detalham estes dois momentos e aprofundam os mecanismos usados para manter a compatibilidade da API e controlar a comunicação através da fronteira de isolamento.

5.2 Arquitetura em Build-time

O momento de construção é o ponto em que a solução transforma uma dependência npm comum numa dependência isolada. Esta opção evita que a aplicação hospedeira tenha de invocar manualmente uma API de isolamento ou alterar as suas importações. Em vez disso, o desenvolvedor declara, na Configuração da *Sandbox*, quais as bibliotecas que devem ser isoladas e que capacidades lhes podem ser expostas. A partir dessa declaração, o processo de construção produz dois artefactos complementares: um *proxy* que passa a ser importado pela aplicação e um bundle separado que contém a biblioteca real a executar no ambiente isolado.

A escolha do Webpack como ponto de integração decorre do papel que este já desempenha na construção da aplicação. O bundler conhece o grafo de dependências, resolve importações de pacotes npm e permite substituir módulos durante a compilação (Webpack 2026b). Assim, a solução pode interceder apenas as bibliotecas configuradas para isolamento e redirecionar essas importações para *proxies* gerados, sem modificar o código fonte aplicacional. Esta decisão preserva a ergonomia de utilização ao permitir que a aplicação continue a declarar, por exemplo, uma importação da biblioteca original, embora o módulo efetivamente incluído no bundle da aplicação passe a ser o *proxy*.

Esta intercepção é realizada a partir da configuração declarativa da *sandbox*. Durante a aplicação do plugin, a configuração é carregada e o conjunto de bibliotecas a isolar é convertido em regras de substituição de módulos. Para cada biblioteca configurada, o plugin associa o nome do pacote a um ficheiro de *proxy*. Quando o Webpack resolve uma importação que coincide com esse nome, a resolução é desviada para o *proxy* correspondente.

O *proxy* é gerado automaticamente a partir da superfície exportada pela biblioteca. Para esse efeito, a solução consulta os ficheiros de declaração TypeScript (`.d.ts`) da biblioteca ou, quando necessário, os ficheiros do pacote `@types` correspondente. A TypeScript Compiler API é usada para construir uma representação estruturada das exportações disponíveis, evitando uma análise textual mais propensa a erros das declarações (Microsoft 2026b). Esta decisão é distinta da abordagem baseada em interfaces WIT discutida na Secção 3.2: no processo de construção desta solução, as declarações TypeScript não são usadas para traduzir toda a semântica da biblioteca para um contrato estático de baixo nível, mas para identificar os nomes que o *proxy* deve reexportar. A preservação do comportamento observável da API é tratada posteriormente, em tempo de execução, através do redirecionamento das chamadas para objetos que permanecem no ambiente isolado, complementado pelos mecanismos de marshalling apenas quando valores concretos precisam de atravessar a fronteira.

O resultado desta fase é um módulo *proxy* que reproduz a forma de importação esperada pela aplicação. Para exportações funcionais ou construtíveis, o *proxy* encaminha chamadas, instanciações e acessos a propriedades para a API mediada devolvida durante a inicialização da *sandbox*. Para bibliotecas cujo tipo TypeScript usa o padrão `export =`, como ocorre em bibliotecas CommonJS com um singleton invocável, o gerador produz uma exportação padrão compatível com a chamada direta e com o acesso a propriedades. Desta forma, a geração automática procura manter a correspondência entre a superfície pública da biblioteca original e a superfície disponibilizada à aplicação, reduzindo a necessidade de código de adaptação manual.

Em paralelo com a geração do *proxy*, o plugin cria um bundle separado da biblioteca real através de um *child compiler* do Webpack. Este bundle é produzido como um artefacto auxiliar, com a biblioteca empacotada num formato que pode ser avaliado posteriormente

dentro do ambiente QuickJS/Wasm. Esta separação é uma parte central da conceção. Desta forma, a aplicação passa a depender do *proxy*, enquanto o código da dependência isolada fica disponível como entrada para o ambiente isolado. O *child compiler* permite que esta etapa reutilize a resolução e as regras do próprio processo de construção da aplicação, evitando uma invocação externa e separada do Webpack para cada biblioteca (Webpack 2026a).

A Figura 5.4 apresenta a sequência principal deste processo. O desenvolvedor fornece a configuração; o plugin identifica as bibliotecas configuradas; o gerador cria o *proxy* com uma superfície compatível; e as importações da aplicação passam a resolver para esse *proxy*.

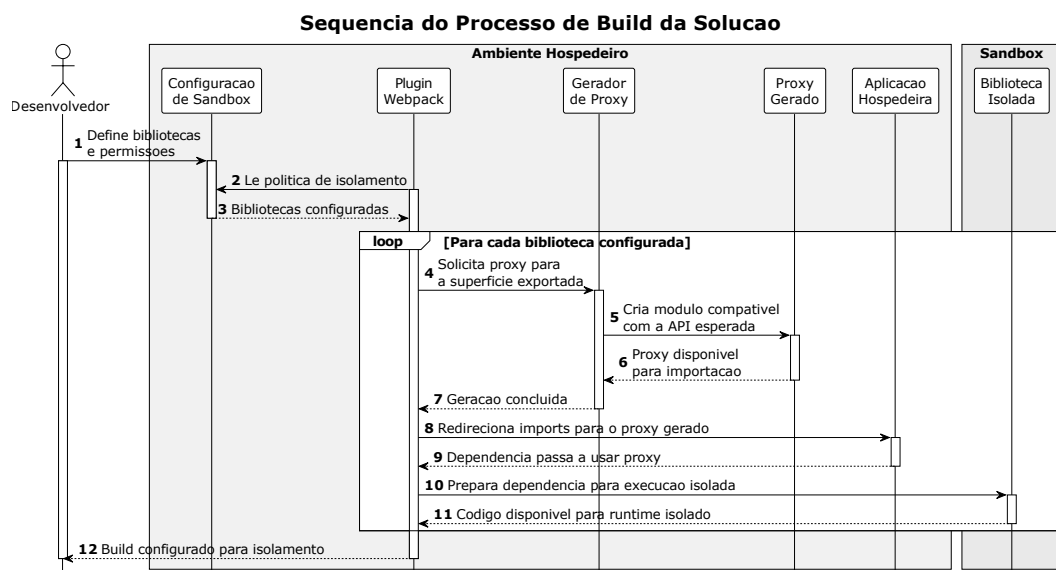


Figura 5.4: Sequência do processo de construção da solução

Esta arquitetura em tempo de construção (*build-time*) tem três benefícios principais. Primeiro, torna a adoção mais transparente, porque a aplicação não precisa de substituir manualmente importações por chamadas explícitas a uma API de isolamento. Segundo, torna a geração dos artefactos repetível a partir da configuração e das declarações da biblioteca. Terceiro, separa o bundle da aplicação do bundle da biblioteca isolada, o que permite que a biblioteca seja carregada e avaliada apenas pelo ambiente isolado.

O plano permite também identificar alguns compromissos. A solução fica dependente do processo de construção e, nesta concretização, da integração com Webpack. Aplicações que usem outro bundler exigiriam uma camada equivalente de interceção de importações, geração de *proxies* e produção de bundles auxiliares. Além disso, o ecossistema npm inclui diferentes formatos de módulos, padrões de exportação e declarações TypeScript incompletas ou inexistentes. Por isso, a geração automática de *proxies* reduz o esforço de integração, mas não elimina a necessidade de tratar casos especiais quando a forma pública da biblioteca não é descrita de modo suficiente pelos seus ficheiros de declaração.

5.3 Arquitetura em Runtime

Depois da fase de construção, a aplicação deixa de importar diretamente a biblioteca configurada e passa a importar o *Proxy Gerado*. A arquitetura em tempo de execução começa

nesse ponto. Quando o módulo *proxy* é inicializado, este carrega o bundle isolado produzido no processo de construção e solicita ao Gestor de *Sandbox* a criação do ambiente de execução para a biblioteca correspondente. A partir desse momento, o código da biblioteca deixa de executar no contexto JavaScript principal da aplicação e passa a executar num contexto QuickJS alojado em Wasm.

Esta decisão concretiza o requisito RF-03, que exige que a biblioteca isolada execute num ambiente separado do contexto principal da aplicação. Com esta separação, cada biblioteca configurada recebe uma instância própria de QuickJS/Wasm e um contexto próprio de execução. Esta opção responde também ao RNF-S02, pois alterações realizadas dentro do ambiente da biblioteca, como mutações no seu estado global ou nos seus protótipos internos, não se propagam diretamente para o ambiente JavaScript da aplicação hospedeira.

A escolha de QuickJS é motivada pela necessidade de executar bibliotecas JavaScript sem as traduzir para outro modelo de programação. Como discutido na Secção 3.2, uma abordagem baseada em interfaces estáticas exigiria modelar previamente a superfície dinâmica das bibliotecas JavaScript e das capacidades do hospedeiro. No conceito adotada, QuickJS permite avaliar o próprio bundle JavaScript da biblioteca dentro de uma VM embutida. O Wasm fornece a fronteira de isolamento de memória que separa essa VM do ambiente hospedeiro, satisfazendo a restrição RST-02 e reforçando o objetivo de reduzir a superfície de ataque definido em RNF-S01.

O Gestor de *Sandbox* centraliza a criação e preparação deste ambiente. Primeiro, obtém uma instância QuickJS/Wasm e cria um novo contexto para a biblioteca. Depois, prepara a camada de comunicação com o hospedeiro através de uma *Arena*, fornecida por `quickjs-emsripten-sync`. Esta camada é relevante para a compatibilidade da solução. A aplicação deve conseguir continuar a invocar funções, construtores e objetos da biblioteca de forma próxima da utilização normal, conforme estabelecido em RF-02 e RNF-U01. Assim, o ambiente isolado expõe uma API mediada, capaz de encaminhar chamadas através da fronteira entre o ambiente hospedeiro e o ambiente isolado.

Antes de avaliar a biblioteca, o gestor consulta a Configuração da *Sandbox* associada à biblioteca e prepara apenas as capacidades autorizadas do ambiente hospedeiro. Esta responsabilidade liga a arquitetura em tempo de execução aos requisitos RF-04, RF-05 e RNF-S03. A biblioteca isolada não recebe acesso implícito ao ambiente global da aplicação. Quando precisa de uma capacidade externa, como uma API nativa do ambiente, essa capacidade tem de estar explicitamente configurada e é exposta através dos mecanismos descritos na Secção 5.4.

Com o contexto preparado, o gestor avalia o bundle da biblioteca dentro da VM. O bundle produzido em tempo de construção disponibiliza as exportações da biblioteca através de uma variável global interna ao ambiente isolado. O gestor obtém essa superfície exportada e aplica uma camada de mediação antes de a devolver ao *proxy*. O *proxy*, por sua vez, reexporta esses valores para a aplicação. Do ponto de vista da aplicação, uma chamada à biblioteca continua a parecer uma chamada normal. No entanto, arquiteturalmente, essa chamada passa pelo *proxy* e pelos objetos mediados devolvidos durante a inicialização. A operação é então executada dentro do contexto QuickJS/Wasm, sem que o Gestor de *Sandbox* tenha de atuar como intermediário em cada invocação normal.

A Figura 5.5 apresenta a sequência principal deste processo. A vista de processos de nível três evidencia que o *Proxy Gerado* é o ponto de entrada para a aplicação, enquanto o Gestor de *Sandbox* cria o ambiente isolado, aplica a política de permissões, prepara a Membrana

Host-VM para capacidades autorizadas e usa a Membrana VM-Host para devolver uma API mediada ao *proxy*. Depois dessa preparação, as chamadas da aplicação seguem pela Membrana VM-Host, preservando a ligação aos objetos reais mantidos dentro da VM.

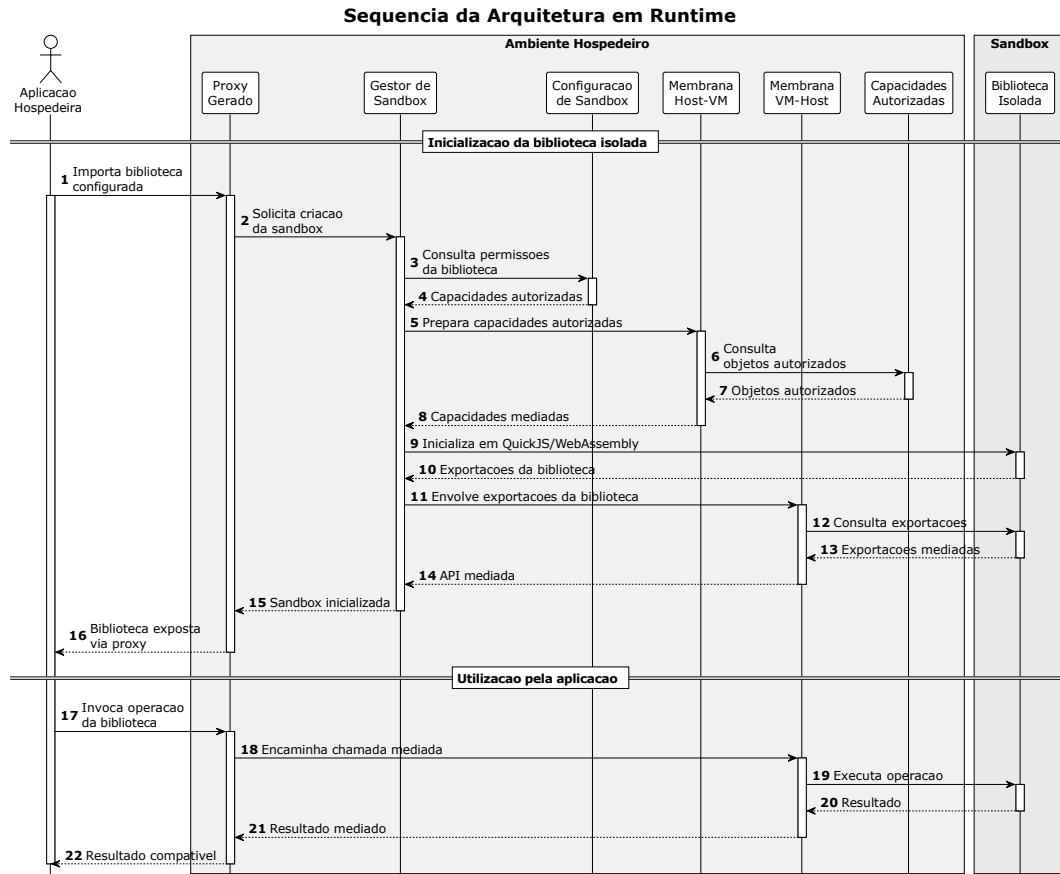


Figura 5.5: Sequência de inicialização e utilização da arquitetura em runtime

A criação de um contexto isolado por biblioteca é uma das decisões de maior relevância. Uma alternativa possível seria reutilizar uma única instância QuickJS para várias bibliotecas, reduzindo custos de inicialização e memória. No entanto, essa opção enfraqueceria a granularidade do isolamento, porque bibliotecas distintas poderiam partilhar estado, objetos ou referências dentro do mesmo ambiente. A solução privilegia, portanto, o isolamento por biblioteca, alinhado com RF-01 e RF-03, aceitando como custo uma maior utilização de memória e tempo adicional de inicialização. Este compromisso é coerente com RNF-P01, onde a solução não procura eliminar todo o sobrecusto, mas sim manter um equilíbrio defensável entre segurança, compatibilidade e desempenho.

Esta arquitetura também delimita o efeito de uma biblioteca comprometida. Se uma dependência tentar alterar o seu estado global, redefinir protótipos ou conservar referências internas maliciosas, esses efeitos ficam confinados ao seu contexto QuickJS/Wasm e às capacidades explicitamente expostas. A aplicação continua a receber resultados por meio do *proxy*, mas não partilha diretamente o *heap* da biblioteca. Desta forma, a arquitetura em tempo de execução materializa o princípio de menor privilégio definido no Capítulo 4 e prepara a base para o modelo de permissões e membranas descrito na Secção seguinte.

5.4 Modelo de Permissões e Membranas

A criação de um contexto QuickJS/Wasm por biblioteca estabelece a fronteira de isolamento, mas não resolve por si só a comunicação entre a biblioteca isolada e a aplicação hospedeira. Uma biblioteca JavaScript precisa de receber argumentos, devolver resultados e, em alguns casos, usar capacidades do ambiente hospedeiro. O modelo de permissões e membranas responde a este problema: permite a comunicação necessária para preservar compatibilidade, mas impede que a biblioteca isolada herde automaticamente a autoridade do ambiente hospedeiro.

O modelo de permissões é granular por biblioteca. A configuração da *sandbox* associa cada dependência a uma lista explícita de capacidades autorizadas, mantendo a política de isolamento fora do código aplicacional e num formato simples de ler e alterar. O Excerto 5.1 apresenta um exemplo reduzido desta estrutura.

```
1 export default {  
2   "image-js": {  
3     permissions: ["Intl", "TextEncoder", "TextDecoder"],  
4   },  
5   "date-fns": {  
6     permissions: ["Intl"],  
7   },  
8   "marked": {  
9     permissions: [],  
10  },  
11 };
```

Listing 5.1: Exemplo declarativo de permissões por biblioteca

Neste modelo, a ausência de uma capacidade na lista de permissões deve ser interpretada como negação, não como autorização implícita. Durante a criação da *sandbox*, cada nome declarado em `permissions` é resolvido no ambiente hospedeiro através de `globalThis`. Se a capacidade existir e estiver declarada para aquela biblioteca, é adaptada pela membrana antes de ser exposta ao contexto QuickJS/Wasm. Se não estiver declarada, não é disponibilizada à biblioteca isolada. Esta decisão concretiza RF-04 e RF-05, pois permite selecionar as capacidades acessíveis por biblioteca, e responde ao RNF-S03, ao aplicar o princípio do menor privilégio. O Anexo A apresenta uma seleção de capacidades Web documentadas pela Mozilla Developer Network (MDN) que são particularmente relevantes para este modelo de exposição controlada, organizadas por categoria e relevância de segurança.

A Tabela 5.1 sintetiza as capacidades usadas no exemplo anterior. A tabela torna explícito que uma biblioteca pode receber apenas as capacidades de que necessita, ou nenhuma capacidade global adicional quando a sua execução não depende do ambiente hospedeiro.

Tabela 5.1: Capacidades ilustradas no exemplo de configuração

Capacidade	Objeto global exposto	Bibliotecas no exemplo	Observação de segurança
Intl	APIs de internacionalização, formatação e localização disponibilizadas pelo ambiente hospedeiro	image-js, date-fns	Exposta apenas a bibliotecas que dependem de formatação ou comportamento dependente de locale.
TextEncoder	Codificação de texto em sequências de bytes	image-js	Concede uma capacidade funcional específica, sem expor outras APIs globais do ambiente hospedeiro.
TextDecoder	Decodificação de sequências de bytes em texto	image-js	Complementa a capacidade de codificação quando a biblioteca precisa de converter dados binários e textuais.
Sem permissões	Nenhuma capacidade global adicional	marked	A biblioteca executa no contexto isolado sem acesso declarado a capacidades globais do ambiente hospedeiro.

A fronteira host→VM inclui uma membrana que adapta a capacidade antes da exposição ao contexto isolado. Esta decisão é necessária porque várias APIs nativas do ambiente JavaScript dependem de estado interno e de um recetor correto durante a invocação (*ECMAScript® 2027 Language Specification* 2026). Se uma API nativa for usada através de um *proxy* genérico, certas chamadas podem falhar por incompatibilidade do recetor ou produzir comportamento diferente do esperado. A membrana reduz esse problema ao apresentar à VM uma referência mediada que redireciona as chamadas para o objeto original do ambiente hospedeiro, sem entregar acesso direto.

A Figura 5.6 mostra o fluxo de acesso autorizado a uma capacidade do ambiente hospedeiro. O ponto essencial é que a biblioteca nunca consulta diretamente o ambiente global. Durante a criação da *sandbox*, o gestor consulta a configuração, envolve a capacidade autorizada através da membrana host→VM e expõe a referência resultante no contexto QuickJS. Durante a utilização, o código da biblioteca invoca essa referência dentro do próprio ambiente isolado, a chamada atravessa então a membrana e só depois alcança a capacidade autorizada no ambiente hospedeiro.

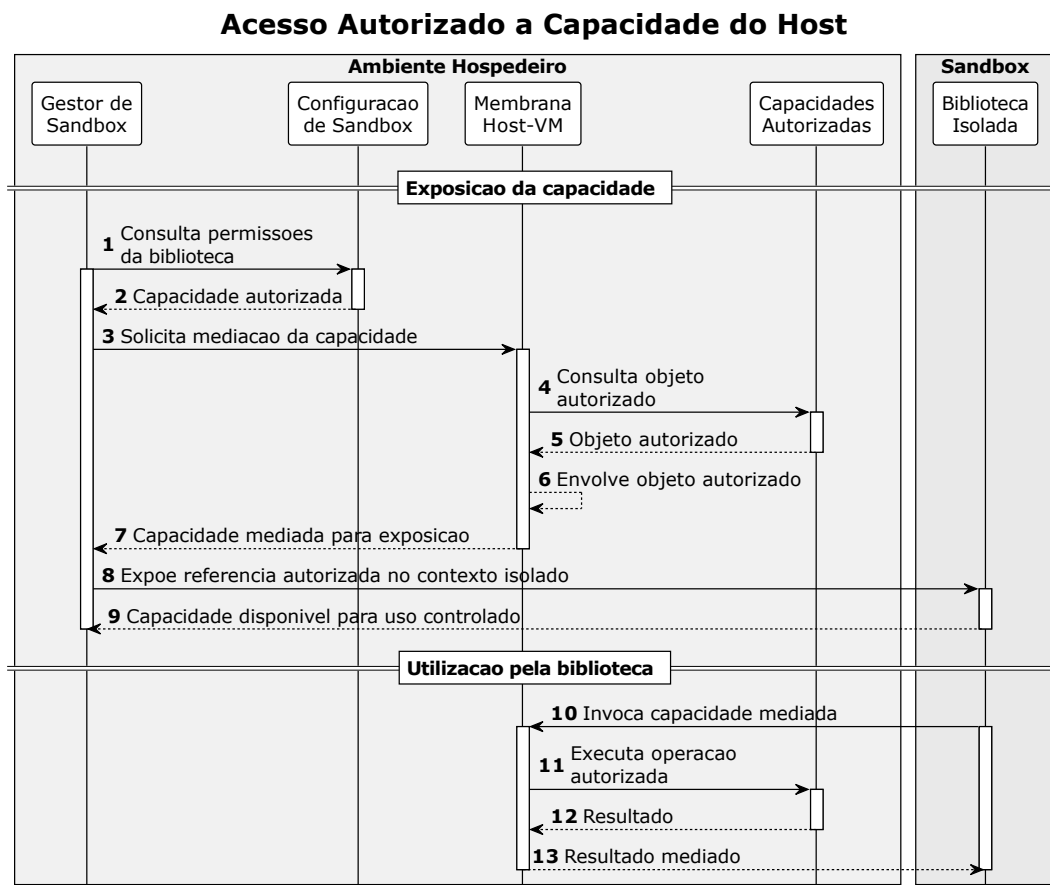


Figura 5.6: Acesso autorizado a uma capacidade do ambiente hospedeiro

Esta direção da membrana contribui para RNF-S01 e RNF-S02. A superfície de ataque é reduzida porque apenas capacidades configuradas são expostas. Além disso, a biblioteca não tem acesso direto ao ambiente global. Um ataque baseado em acesso indevido a APIs do browser fica limitado pela política de permissões, enquanto alterações internas ao contexto isolado permanecem confinadas ao ambiente QuickJS/Wasm.

Na direção VM→host, o problema é preservar a forma de utilização da biblioteca pela aplicação. Bibliotecas JavaScript expõem funções, construtores, objetos com estado e propriedades cujo comportamento depende do contexto onde continuam a existir. Se as exportações fossem convertidas para objetos simples, perder-se-iam propriedades importantes da semântica JavaScript, como o recetor correto de métodos, getters de protótipo ou a identidade de objetos com estado.

Por isso, a solução usa uma membrana VM→host sobre as exportações da biblioteca. O *proxy* apresentado à aplicação contém referências mediadas para valores que permanecem no contexto isolado. Assim, quando a aplicação invoca uma função, construtor ou método, a operação é redirecionada para a biblioteca isolada. Resultados simples, como números ou strings, atravessam a fronteira como valores convertidos. No caso de objetos com comportamento próprio, continuam representados no ambiente hospedeiro por referências mediadas, de modo que interações posteriores sejam executadas sobre o objeto original mantido na VM. Esta decisão materializa RF-02 e apoia RNF-U01, porque preserva a forma

habitual de uso da biblioteca sem exigir que a aplicação seja reestruturada em torno de chamadas explícitas de isolamento.

A Figura 5.7 sintetiza o fluxo de uma chamada da aplicação para uma exportação da biblioteca isolada. A aplicação invoca a operação através do *proxy*; esse *proxy* delega para a referência mediada que recebeu na inicialização; a membrana VM→host executa a chamada no ambiente isolado sobre a exportação real da biblioteca; e o resultado regressa ao ambiente hospedeiro como valor convertido ou como nova referência mediada. Quando o resultado é uma referência mediada, interações posteriores com esse objeto ou função continuam a ser redirecionadas pela membrana para o objeto original mantido na *sandbox*.

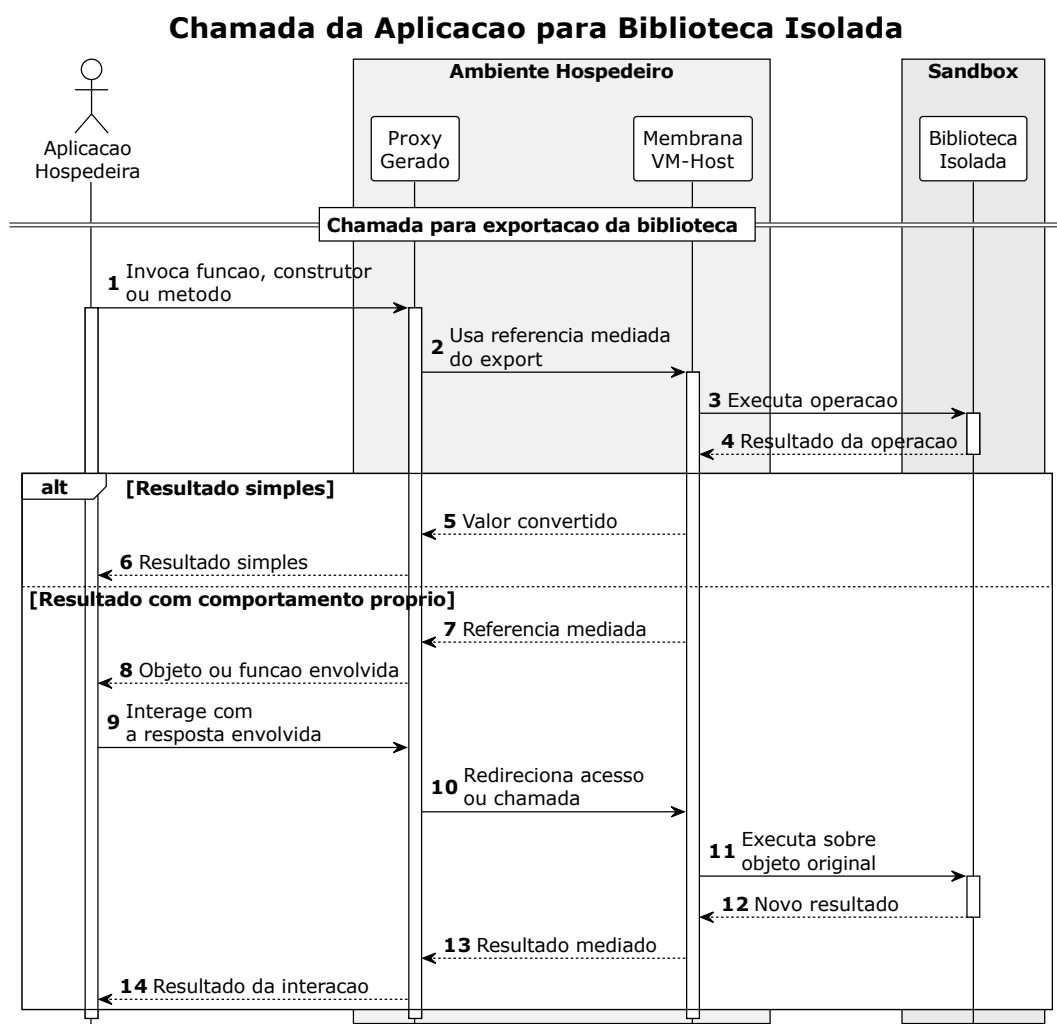


Figura 5.7: Mediação de uma chamada da aplicação para a biblioteca isolada

A base desta comunicação não é apenas a serialização de valores, mas a preservação de identidade e funcionalidade. As exportações da biblioteca e os objetos comportamentais devolvidos por ela continuam a existir no contexto QuickJS e o ambiente hospedeiro recebe referências que redirecionam chamadas, acessos a propriedades e instanciações para esses objetos. O marshalling continua a ser necessário, mas para a conversão de objetos de menor complexidade que precisam efetivamente de atravessar a fronteira. A arquitetura

deve admitir a extensão do marshalling quando novos padrões de dados exigem tratamento próprio. Esta linha de evolução é retomada na Secção 5.5.

É importante clarificar que a biblioteca isolada não fica impedida de modificar os seus próprios protótipos ou objetos globais dentro do ambiente isolado. O ponto de segurança é que essas alterações ocorrem num módulo QuickJS/Wasm separado e não afetam diretamente os protótipos da aplicação hospedeira. Esta propriedade reduz a propagação de ataques baseados em estado global partilhado, como poluição de protótipos, cumprindo RNF-S02.

O principal custo desta abordagem é a complexidade da fronteira. A membrana tem de equilibrar dois objetivos: preservar compatibilidade suficiente para bibliotecas JavaScript e manter uma fronteira de segurança explícita. Quanto mais transparente for a API exposta à aplicação, maior é a necessidade de tratar casos específicos, como objetos com estado ou tipos nativos. Além disso, cada chamada que atravessa a fronteira implica mediação e conversão de valores, acrescentando custos de desempenho, o que se relaciona diretamente com RNF-P01. A conceção aceita esse custo como consequência do isolamento granular, e reserva otimizações para padrões de utilização com chamadas frequentes para a Secção 5.5.

5.5 Melhorias Futuras e Extensões da Conceção

A conceção apresentada nas secções anteriores define o núcleo da solução: seleção de bibliotecas a isolar, geração de *proxies*, execução da dependência num contexto QuickJS alojado em Wasm e comunicação controlada através de permissões e membranas. Este núcleo privilegia transparência de adoção e isolamento granular. No entanto, a própria existência de uma fronteira entre a aplicação e a *sandbox* introduz limitações que podem motivar extensões futuras, sobretudo em cenários de desempenho e compatibilidade.

A extensão mais relevante é um modo opcional de execução com maior granularidade, orientado para APIs com muita comunicação. No modo transparente, a aplicação continua a invocar diretamente operações da biblioteca através do *proxy*. Esta opção é adequada como comportamento base porque minimiza alterações no código aplicacional e responde diretamente a RF-02 e RNF-U01. Contudo, bibliotecas com cadeias de chamadas, objetos intermediários ou padrões de construção incremental podem atravessar a fronteira de isolamento muitas vezes durante uma única operação lógica. Nesses casos, a penalização de desempenho deve-se principalmente ao número de travessias, conversões de valores e criação de referências mediadas.

Um modo *coarse-grained* pode reduzir esse custo deslocando uma parte explícita do fluxo de uso da biblioteca para dentro da *sandbox*. Em vez de a aplicação hospedeira conduzir passo a passo uma sequência longa de chamadas, o desenvolvedor poderia definir um adaptador de carga de trabalho: uma função agregada que executa a lógica de utilização da biblioteca no contexto QuickJS/Wasm e devolve apenas o resultado final necessário ao ambiente hospedeiro. Assim, objetos intermediários e chamadas encadeadas permanecem no ambiente isolado, e a fronteira é atravessada por um número significativamente menor de entradas e saídas. Esta alternativa preserva o mesmo modelo de isolamento e permissões, mas altera o ponto de integração usado pela aplicação.

Do ponto de vista do desenvolvedor, esta extensão implica três passos explícitos. Primeiro, a lógica sensível ao custo da fronteira é movida para um módulo próprio, onde a função do adaptador de carga de trabalho importa e usa a biblioteca isolada e contém o bloco de

código que deve ser executado dentro da *sandbox*. Segundo, a aplicação passa a importar essa função agregada e a chamá-la no ponto onde antes conduzia diretamente a sequência de chamadas à biblioteca. Terceiro, a Configuração da *Sandbox* passa a declarar esse adaptador de carga de trabalho dentro da configuração da biblioteca correspondente, para que o processo de construção saiba que esse módulo deve ser tratado como uma unidade de execução associada à mesma *sandbox* da dependência.

Esta opção também altera a fase de construção. Para além de gerar o *proxy* transparente da biblioteca e o bundle isolado da dependência, o plugin teria de processar cada carga de trabalho declarada na configuração, criar um bundle próprio para esse código aplicacional agregado e gerar o ponto de entrada que a aplicação importaria. Esse bundle de carga de trabalho não deve criar uma segunda *sandbox* nem carregar uma segunda instância independente da biblioteca. Deve ser preparado para ser avaliado no mesmo contexto QuickJS/Wasm da biblioteca a que pertence, reutilizando a mesma política de permissões e a mesma instância isolada. Desta forma, a aplicação invoca uma função de maior granularidade, enquanto a sequência interna de chamadas à biblioteca ocorre dentro da *sandbox*.

A Figura 5.8 ilustra a diferença de execução face ao modo transparente apresentado anteriormente. A inicialização da carga de trabalho reutiliza a *sandbox* da biblioteca correspondente, sendo o *proxy* da carga de trabalho a solicitar a exportação ao *Proxy Gerado* da biblioteca. Por sua vez, este carrega a carga de trabalho na *sandbox* existente quando necessário e as exportações da carga de trabalho são devolvidas como referências mediadas. Na utilização, a aplicação faz uma chamada agregada ao *proxy* da carga de trabalho; dentro da *sandbox*, a carga de trabalho executa a sequência de chamadas sobre a biblioteca real e apenas o resultado final atravessa a fronteira de volta para o ambiente hospedeiro.

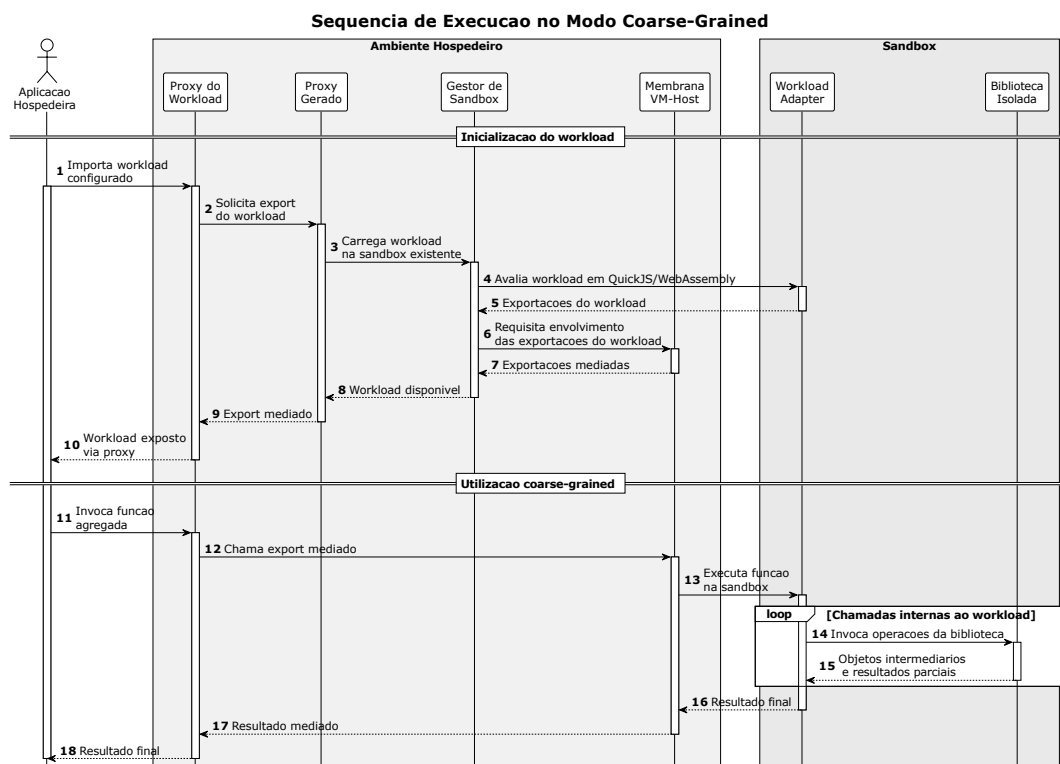


Figura 5.8: Sequência de execução no modo *coarse-grained*

Esta extensão deve permanecer opcional. Torná-la o comportamento principal enfraqueceria de forma contundente a usabilidade da solução. O modo transparente continua a ser a interface predefinida e mais ergonômica. O modo *coarse-grained* deve ser tratado como uma otimização para *hot paths* identificados como sensíveis ao custo das travessias e que podem afetar o desempenho da solução de forma notória. Neste contexto, o compromisso prende-se com menos chamadas através da fronteira que podem melhorar o desempenho, mas exigem que o desenvolvedor torne explícita uma unidade de trabalho que passa a executar dentro da *sandbox*.

Outra linha de evolução diz respeito ao marshalling. A arquitetura depende de conversões entre o ambiente hospedeiro e o contexto QuickJS, mas o ecossistema JavaScript inclui tipos e objetos com semântica própria, estado interno ou comportamento que nem sempre pode ser representado por objetos simples. Do ponto de vista da conceção, isto significa que o mecanismo de marshalling deve ser extensível através de uma interface comum para componentes de marshalling e de um ponto central que os agregue antes da criação de cada ambiente isolado. O objetivo não é transformar a fronteira numa serialização universal, mas permitir que novos padrões de dados sejam suportados de forma controlada quando forem necessários. Esta decisão concretiza o requisito RNF-M02.

O mecanismo concebido separa o Gestor de *Sandbox* dosmarshallers concretos. Quando uma *sandbox* é criada, o gestor obtém primeiro a instância QuickJS/Wasm e o respetivo contexto. Em seguida, solicita a um agregador de marshalling a construção da configuração da Arena para esse ambiente isolado. Esse agregador instancia todos osmarshallers disponíveis, associa-os ao contexto QuickJS recém-criado e devolve ao gestor um conjunto único de opções de comunicação. O gestor usa essas opções ao criar a Arena, sem precisar de conhecer que tipos específicos são suportados por cada marshaller. Assim, todos os ambientes isolados passam a receber o mesmo conjunto de extensões de marshalling de forma uniforme.

Cada marshaller segue a mesma interface conceptual. Deve expor um predicado para valores $host \rightarrow VM$, indicando se um valor deve ser tratado pelo marshaller ou pode seguir pelo mecanismo predefinido; um predicado equivalente para valores $VM \rightarrow host$, permitindo impedir que a Arena transforme prematuramente um valor do ambiente isolado num *proxy* genérico; uma operação de conversão $host \rightarrow VM$; e uma operação de conversão $VM \rightarrow host$. As operações de conversão só produzem resultado quando o valor pertence ao tipo tratado por esse marshaller, caso contrário, deixam o valor seguir para os restantes mecanismos da Arena. Esta composição permite que váriosmarshallers coexistam sem que a lógica de decisão fique espalhada pelo Gestor de *Sandbox*.

Para adicionar suporte a um novo tipo, o ponto de extensão previsto é a pasta demarshallers. O desenvolvedor deve criar um novo ficheiro que implemente a interface comum e disponibilize uma função de criação de marshaller para um ambiente isolado. Essa função é depois incluída no conjunto agregado demarshallers disponibilizado aos ambientes isolados, mantendo o Gestor de *Sandbox* inalterado. Desta forma, o esforço de extensão fica localizado no novo módulo de marshalling e no registo central que compõe a configuração da Arena.

5.6 Síntese das Decisões Arquiteturais

As decisões descritas ao longo do capítulo procuram equilibrar isolamento, compatibilidade, desempenho e usabilidade. A solução não isola a aplicação completa, mas sim bibliotecas

selecionadas, mantendo a integração com o fluxo normal de desenvolvimento e tornando explícitas as capacidades concedidas a cada dependência.

A Tabela 5.2 sintetiza as principais decisões de concepção e a sua relação com os requisitos definidos no Capítulo 4.

Tabela 5.2: Síntese das decisões arquiteturais de concepção

Decisão arquitetural	Requisitos associados
Isolamento por biblioteca em contexto QuickJS/Wasm dedicado	RF-01, RF-03, RNF-S01, RNF-S02, RST-02
Configuração declarativa de <i>sandbox</i> e permissões por biblioteca	RF-04, RF-05, RNF-S03, RNF-U02
Integração no processo de construção através de Webpack e redirecionamento de importações	RF-01, RF-02, RNF-U01
Geração automática de <i>proxies</i> a partir da superfície exportada da biblioteca	RF-02, RNF-U01, RNF-M01
Separação entre bundle da aplicação e bundle da biblioteca isolada	RF-03, RST-03
Membranas <i>host</i> → <i>VM</i> e <i>VM</i> → <i>host</i> para comunicação e exposição controlada	RF-02, RF-04, RF-05, RNF-S01, RNF-S03
Mecanismo extensível de marshalling para tipos com tratamento específico	RNF-M02
Modo <i>coarse-grained</i> como extensão opcional para cargas de trabalho sensíveis ao custo da fronteira	RNF-P01, RNF-M01

Capítulo 6

Implementação

O capítulo anterior definiu a arquitetura da solução em termos de componentes, fronteiras e fluxos principais. Este capítulo descreve como essa arquitetura foi concretizada no protótipo desenvolvido. A descrição segue a mesma separação conceptual entre processo de construção e tempo de execução, mas aprofunda os mecanismos técnicos que ficaram abstraídos na conceção: utilização das APIs do Webpack, análise de declarações TypeScript, geração dos *proxies*, inicialização de QuickJS/Wasm, aplicação das membranas e composição dos marshallers.

A implementação tem como comportamento base o modo transparente. Neste modo, a aplicação continua a importar e a usar a biblioteca JavaScript pela interface habitual, enquanto o processo de construção redireciona essa importação para um *proxy* gerado. O modo *coarse-grained*, baseado em adaptadores de carga de trabalho, é tratado apenas na Secção 6.5, porque constitui uma extensão opcional para caminhos de execução com muitas travessias da fronteira.

6.1 Implementação do Plugin do Processo de Construção

O plugin do processo de construção concretiza a parte da arquitetura responsável por transformar uma biblioteca comum numa biblioteca isolada. O plugin recebe a Configuração da *Sandbox*, identifica as bibliotecas declaradas e prepara dois artefactos por biblioteca: um *proxy* importado pela aplicação e um bundle separado com a implementação real da dependência.

A configuração é o ponto de partida do processo. No modo transparente, cada entrada associa o nome da biblioteca a uma lista explícita de permissões. A ausência de uma permissão mantém a capacidade indisponível por omissão. O Excerto 6.1 mostra um exemplo simplificado para uma biblioteca que necessita de acesso a `Intl`.

```
1 export default {  
2   "date-fns": {  
3     permissions: ["Intl"],  
4   },  
5 };
```

Listing 6.1: Exemplo de configuração de uma biblioteca no modo transparente

Durante a aplicação do plugin, a configuração é carregada a partir do contexto do compilador. Em seguida, os nomes das bibliotecas são obtidos com `Object.keys`. Desta forma, a

chave de cada entrada é simultaneamente o nome do pacote npm a interceptar e o identificador usado pelo *runtime* para consultar permissões.

A interceção de importações é feita com `NormalModuleReplacementPlugin` (Webpack 2026c). Para cada biblioteca configurada, o plugin cria uma regra que corresponde à importação exata do pacote e substitui o pedido pelo módulo *proxy* gerado para essa biblioteca. O Excerto 6.2 mostra a forma essencial desta substituição.

```
1 new NormalModuleReplacementPlugin(  
2   new RegExp('^${libName}$'),  
3   (resource) => {  
4     resource.request = proxyPath;  
5   },  
6 ).apply(compiler);
```

Listing 6.2: Redirecionamento de importações para o *proxy* gerado

Assim, o código da aplicação pode continuar a escrever, por exemplo, `import { format } from "date-fns"`, e o módulo efetivamente incluído no bundle principal passa a ser o *proxy*, não a biblioteca original. Isto concretiza a transparência descrita na conceção sem exigir alterações manuais nos pontos de importação da aplicação.

O plugin associa a geração dos *proxies* ao *hook compile*, um dos hooks do compilador documentados pelo Webpack (Webpack 2026b). Na implementação desenvolvida, esta associação é feita no compilador principal, evitando que a geração seja executada novamente nos *child compilers* criados mais tarde para produzir os bundles isolados. A fase de geração é, portanto, separada da fase de empacotamento da biblioteca. Na primeira, o plugin escreve o módulo *proxy*; na segunda, cria o bundle que será avaliado dentro do ambiente isolado.

O bundle isolado é produzido através de um *child compiler* do Webpack. A implementação resolve a entrada real da biblioteca com `require.resolve`, cria um compilador filho com `compilation.createChildCompiler` e define a saída como uma biblioteca JavaScript do tipo `var`. O nome global usado é `__SANDBOXED_LIB__`, que será lido pelo Gestor da *Sandbox* durante a avaliação do bundle. O Excerto 6.3 resume esta configuração.

```
1 const outputOptions = {  
2   filename: `${libName}.bundle.js`,  
3   path: path.resolve(compiler.options.output.path || "dist"),  
4   library: {  
5     type: "var",  
6     name: "__SANDBOXED_LIB__",  
7   },  
8 };  
9  
10 const childCompiler = compilation.createChildCompiler(  
11   `SandboxCompiler-${libName}`,  
12   outputOptions,  
13   [new EnableLibraryPlugin("var")],  
14 );
```

Listing 6.3: Criação do bundle isolado da biblioteca

O uso de *child compilers* permite reutilizar a resolução e a configuração do próprio Webpack, em vez de invocar um processo externo por biblioteca. Ao mesmo tempo, mantém o bundle da aplicação separado do bundle da dependência isolada. Esta separação é necessária

porque a aplicação deve importar o *proxy*, enquanto a implementação da biblioteca deve ser carregada e avaliada apenas pelo ambiente isolado, tal como descrito na Secção 5.2.

6.2 Geração e Conteúdo dos Proxies

O Gerador de *Proxies* concretiza a superfície compatível descrita na concepção. A implementação está dividida entre o analisador sintático de declarações TypeScript e o gerador textual do módulo *proxy*. O objetivo é descobrir que nomes devem ser reexportados e produzir código que encaminhe chamadas, acessos e instanciações para a API mediada devolvida pela *sandbox*.

A descoberta de exportações usa a TypeScript Compiler API. O plugin procura declarações `.d.ts` dentro do pacote instalado. Quando a biblioteca não fornece declarações próprias, tenta usar o pacote correspondente em `@types`. O analisador sintático cria um `Program`, obtém o `TypeChecker` e consulta os símbolos exportados das declarações encontradas. Esta abordagem nativa evita depender de expressões regulares sobre texto TypeScript e permite lidar com declarações distribuídas por vários módulos.

O analisador sintático trata também bibliotecas que usam o padrão `export =`. Este caso foi descoberto ao testar a solução com bibliotecas CommonJS que expõem um singleton invocável, como acontece com `lodash`. Nesses casos, os nomes devolvidos por `getExportsOfModule` podem corresponder a membros de espaço de nomes ao nível de tipos, e não a exportações reais em tempo de execução. Por isso, quando o analisador sintático deteta uma atribuição `export = identificador`, regista esse identificador como `callableSingletonExport` e evita gerar reexportações nomeadas que não existiriam no objeto real.

O conteúdo gerado para cada *proxy* tem uma estrutura estável. Primeiro, o *proxy* carrega o bundle isolado produzido pelo *child compiler*. Para permitir a execução dos testes de compatibilidade descritos na Secção 7.1, o *proxy* usa `readFileSync` em ambientes Node.js e `fetch` no browser. Depois, usa o Gestor de *Sandbox* para criar o ambiente isolado através da função `createSandbox`. Em seguida, obtém `librarySandbox.exports`, que corresponde à superfície da biblioteca já envolvida pela Membrana VM-Host. Por fim, reexporta cada elemento da API esperada pela aplicação.

```
1 let libraryCode;
2 if (typeof process !== "undefined" && process.versions?.node) {
3   const { readFileSync } = await import("fs");
4   libraryCode = readFileSync(bundlePath, "utf-8");
5 } else {
6   libraryCode = await fetch(bundleUrl).then((r) => r.text());
7 }
8
9 const librarySandbox = await SandboxManager.createSandbox(
10   "date-fns",
11   libraryCode
12 );
13
14 const proxyExports = librarySandbox.exports;
```

Listing 6.4: Estrutura inicial de um *proxy* gerado

Para exportações funcionais ou construtíveis, o *proxy* gerado usa um Proxy JavaScript com traps para chamada, instanciação e acesso a propriedades (*Proxy - JavaScript | MDN*

2024). A trap `apply` encaminha chamadas normais para a função mediada pela Membrana VM-Host. A trap `get` preserva o acesso a métodos ou propriedades estáticas. Já a trap `construct` garante que a instância é construída diretamente dentro do ambiente isolado, onde se encontram o construtor real, o protótipo, e qualquer estado interno da biblioteca. Por isso, a trap delega a operação para `librarySandbox.createInstance`. O Excerto 6.5 mostra uma versão reduzida deste padrão.

```

1 function _DateTime_target() {}
2 const _export_DateTime = typeof proxyExports["DateTime"] === "function"
3 ? new Proxy(_DateTime_target, {
4   apply(_target, _thisArg, args) {
5     return proxyExports["DateTime"](...args);
6   },
7   construct(_target, args) {
8     return librarySandbox.createInstance("DateTime", args);
9   },
10  get(_target, key, receiver) {
11    if (typeof key === "symbol") {
12      return Reflect.get(proxyExports["DateTime"], key, receiver);
13    }
14    return proxyExports["DateTime"][key];
15  }
16 })
17 : proxyExports["DateTime"];

```

Listing 6.5: Exportação nomeada gerada com suporte a chamada e construção

A distinção entre chamada e instanciação evita uma perda de semântica. Uma chamada comum pode atravessar a membrana como invocação de função e devolver um valor ou uma referência mediada. Uma instanciação, pelo contrário, cria um objeto cujo comportamento posterior pode depender do `this` original, da cadeia de protótipos e de estado interno inicializado pelo construtor. Se o *proxy* invocasse o construtor, recebesse o resultado no ambiente hospedeiro e só depois tentasse guardá-lo como referência viva no ambiente isolado, a instância já teria atravessado a fronteira e poderia ser representada por um *proxy* que não corresponde, de forma estável, ao objeto original. A função `createInstance` evita essa ida e volta: cria a instância em QuickJS, guarda-a imediatamente num mapa temporário do ambiente isolado, e só depois devolve ao ambiente hospedeiro uma referência mediada que encaminha chamadas futuras para essa instância.

A geração de exportações `default` segue a mesma lógica, mas inclui uma adaptação adicional para bibliotecas cujo `default` pode ser um singleton. Nessa situação, o gerador pode recorrer a uma exportação nomeada como solução alternativa construtível, preservando simultaneamente acesso a métodos do singleton. Para bibliotecas `export =`, o *proxy* produz uma exportação `default` invocável, com suporte a chamada direta, acesso a propriedades e construção.

6.3 Implementação do Runtime da sandbox

O *runtime* da *sandbox* concretiza a execução isolada da biblioteca depois de o *proxy* ter sido importado. A implementação central está no Gestor de *Sandbox*. O gestor cria o contexto QuickJS/Wasm, prepara a comunicação com o ambiente hospedeiro, avalia o bundle da biblioteca e devolve ao *proxy* uma API mediada.

A criação ocorre pela função `createSandbox`, que usa `newQuickJSAsyncWasmModule`, da biblioteca `quickjs-emsripten`, para criar um módulo Wasm com uma instância QuickJS própria. Esta biblioteca fornece as primitivas de baixo nível do interpretador: criação de módulos QuickJS, criação de contextos com `newContext`, avaliação de código, gestão explícita de *handles*, exposição manual de funções e execução de tarefas pendentes. Desta forma, `quickjs-emsripten` é a camada que disponibiliza o interpretador QuickJS compilado para Wasm, mas não fornece, por si só, uma superfície transparente para passar objetos JavaScript complexos entre ambiente hospedeiro e isolado.

A comunicação de valores entre os ambientes é facilitada pelo objeto *Arena*, da biblioteca `quickjs-emsripten-sync`. Esta biblioteca envolve `quickjs-emsripten` e reduz a necessidade de manipular *handles* manualmente. A *Arena* avalia código com `arena.evalCode`, converte valores entre os dois ambientes, gere a libertação dos *handles* que cria, expõe objetos do ambiente hospedeiro como globais do ambiente isolado com `arena.expose` e permite sincronizar objetos através de `arena.sync`. A implementação usa esta camada como ponte base, mas fornece-lhe uma configuração própria através de `createArenaConfig`. Essa configuração define que objetos podem ser envolvidos por *proxies*, que *handles* do ambiente isolado podem ser envolvidos e que *marshallers* personalizados devem ser usados para certos tipos.

No entanto, o *marshalling* disponibilizado pela *Arena* não deve ser confundido com a preservação direta de uma instância viva no outro contexto. Quando é usada uma conversão estrutural completa, a biblioteca cria objetos correspondentes no lado de destino, copia protótipos e descritores de propriedades, e converte funções em *wrappers* que voltam a chamar a função original através da fronteira (Re:Earth 2026). Este comportamento é útil para muitos valores JavaScript, mas continua a depender de reconstrução, *proxies* e nova conversão do recetor quando uma função é invocada. Por isso, a *Arena* é suficiente como infraestrutura de passagem de valores e *handles*, mas não substitui, por si só, uma membrana desenhada para manter a identidade de objetos comportamentais criados dentro do ambiente isolado.

A relação entre as duas bibliotecas é, portanto, hierárquica. `quickjs-emsripten` cria o ambiente QuickJS/Wasm e oferece as operações fundamentais sobre contextos e *handles*. `quickjs-emsripten-sync` usa esse contexto para construir uma abstração de maior nível, a *Arena*, que automatiza conversão, sincronização e exposição de objetos. O código implementado neste trabalho usa a *Arena* para evitar a maior parte do trabalho manual de *marshalling*, mas acrescenta membranas e *marshallers* personalizados porque o *marshalling* genérico da *Arena* não é suficiente para permitir a compatibilidade pretendida com a utilização das bibliotecas JavaScript.

Depois de criada a *Arena*, o *runtime* expõe as capacidades autorizadas, avalia o bundle da biblioteca e obtém a variável global `__SANDBOXED_LIB__`. O Excerto 6.6 resume a sequência principal.

```

1 const { arenaOptions, dispose: disposeMarshallers } =
2   createArenaConfig(vm);
3 const arena = new Arena(vm, arenaOptions);
4
5 exposeWebAPIs(arena, libraryName);
6
7 const libExports = arena.evalCode(
8   `${libraryCode}; __SANDBOXED_LIB__`;
9 );
10
11 const boundExports = bindSandboxExport(
12   arena,
13   libExports,
14   "__SANDBOXED_LIB__",
15   sbHelpers,
16 );

```

Listing 6.6: Sequência principal de inicialização da *sandbox*

A avaliação do bundle ocorre dentro do contexto QuickJS. O bundle foi produzido para deixar a superfície exportada na variável `__SANDBOXED_LIB__`. Por isso, o gestor avalia o código da biblioteca e devolve essa variável como resultado da expressão. Ao invés de ser exposto diretamente, o objeto passa por `bindSandboxExport`, que constrói a Membrana VM-Host sobre funções, classes e objetos exportados. Este mecanismo é descrito em detalhe na Subsecção 6.4.3.

O *runtime* também cria funções auxiliares pré-compiladas dentro do ambiente isolado. Estas funções mantêm um mapa `__SANDBOX_TEMP__` para objetos que devem permanecer no contexto QuickJS e ser chamados posteriormente pelo ambiente hospedeiro através de referências mediadas, mantendo a identidade. A implementação compila funções para guardar objetos, chamar métodos, ler propriedades e construir instâncias. Isto evita recorrer a `arena.evalCode` em cada interação, evitando a sobrecarga de múltiplas compilações do mesmo código e trazendo benefícios de desempenho. Mais importante, permite que a membrana execute operações sobre o objeto vivo no ambiente isolado, em vez de depender de uma cópia materializada no ambiente hospedeiro.

```

1 arena.evalCode("globalThis.__SANDBOX_TEMP__ = new Map();");
2 const sbHelpers = {
3   tempSet: arena.evalCode(
4     "(id, v) => globalThis.__SANDBOX_TEMP__.set(id, v)"
5   ),
6   tempCall: arena.evalCode(
7     "(id, key, ...args) => globalThis.__SANDBOX_TEMP__.get(id)[key](...args)"
8   ),
9   tempGet: arena.evalCode(
10    "(id, key) => globalThis.__SANDBOX_TEMP__.get(id)[key]"
11  ),
12  tempCreate: arena.evalCode(
13    "(ctorKey, id, ...args) => { " +
14    "const inst = new __SANDBOXED_LIB__[ctorKey](...args); " +
15    "globalThis.__SANDBOX_TEMP__.set(id, inst); return inst; }"
16  ),
17 };

```

Listing 6.7: Funções auxiliares pré-compiladas usadas pela Membrana VM-Host

Por fim, o gestor mantém um ciclo para terminar tarefas assíncronas QuickJS. O intervalo verifica `vm.runtime.hasPendingJob()` e executa tarefas pendentes através da Arena. Este mecanismo é necessário porque Promises criadas dentro de QuickJS não são resolvidas automaticamente pelo ciclo de eventos do ambiente hospedeiro. A implementação usa esse controlo para manter a *sandbox* funcional quando é executado trabalho assíncrono dentro do ambiente isolado.

6.4 Permissões, Membranas e Marshallers

A fronteira entre ambientes é implementada por três mecanismos complementares. O primeiro aplica a política de permissões e decide que capacidades do ambiente hospedeiro são expostas. O segundo envolve essas capacidades antes de as sincronizar com QuickJS. O terceiro envolve as exportações da biblioteca antes de as devolver à aplicação. Os marshallers completam esta fronteira quando determinados tipos precisam de conversão específica em vez de um *proxy* genérico. A Arena fornece o canal de passagem de valores, mas a política de exposição, a preservação de identidade e a escolha entre *proxy* e conversão por valor são decisões do projeto.

6.4.1 Exposição de Capacidades Autorizadas

A função `exposeWebAPIs` lê a entrada da biblioteca na configuração e percorre a lista de permissões. Para cada nome autorizado, consulta `globalThis`. Se a capacidade existir no ambiente hospedeiro, é primeiro envolvida pela Membrana Host-VM e só depois sincronizada com `arena.sync`. O valor devolvido por `arena.sync` é o objeto que deve ser entregue ao ambiente isolado através de `arena.expose`, porque é esse *wrapper* que permite sincronizar operações entre os dois contextos. Capacidades ausentes da configuração não são expostas.

Esta implementação concretiza a política de negação por omissão descrita no Capítulo 5. A biblioteca isolada recebe apenas os valores explicitamente listados na sua configuração. O gestor expõe esses valores antes da avaliação do bundle, para que a biblioteca encontre apenas a superfície autorizada durante a execução.

6.4.2 Membrana Host-VM

O problema tratado pela Membrana Host-VM é específico de objetos nativos do ambiente JavaScript. Muitos destes objetos dependem de *internal slots* e validam o valor de `this`. Isto acontece, por exemplo, em objetos de internacionalização, armazenamento, criptografia, consola ou outros objetos Web. Se um método nativo for chamado com um *proxy* genérico como recetor, a chamada pode falhar mesmo quando a capacidade foi autorizada.

Para evitar esse problema, `bindWebApi` constrói um objeto seguro para exposição antes de a Arena o sincronizar. A função percorre a cadeia de protótipos do objeto autorizado, recolhe propriedades próprias e herdadas, liga métodos ao objeto nativo original e envolve objetos aninhados. Esta preparação é necessária porque `arena.sync` consegue criar uma representação sincronizada, mas não sabe, por si só, quais os métodos nativos que devem permanecer ligados ao objeto original para manter os seus *internal slots*. Um `WeakMap` preserva identidade e evita ciclos durante este envolvimento. O Excerto 6.8 mostra a parte central deste comportamento.

```
1 if (typeof apiObject[key] === "function") {  
2   boundObject[key] = makeCallable(apiObject[key], apiObject);  
3 } else {  
4   Object.defineProperty(boundObject, key, {  
5     get: () => bindWebApi(apiObject[key]),  
6     set: (val) => {  
7       apiObject[key] = val;  
8     },  
9   });  
10 }
```

Listing 6.8: Ligação de métodos nativos antes da exposição ao ambiente isolado

A função `makeCallable` cobre ainda o caso de construtores nativos. A Arena pode atravessar funções como chamadas normais, no entanto, algumas APIs exigem `new`. Quando a chamada direta produz um erro compatível com esse padrão, a implementação tenta `Reflect.construct`. Desta forma, capacidades como construtores ou objetos nativos podem ser usadas pela biblioteca isolada sem expor diretamente o objeto original nem entregar ao ambiente isolado uma referência ampla ao ambiente hospedeiro.

6.4.3 Membrana VM-Host

No caso da Membrana VM-Host, o objetivo é preservar a forma de utilização da biblioteca pela aplicação. As exportações reais permanecem dentro do ambiente isolado e o ambiente hospedeiro recebe *proxies* que encaminham chamadas para os objetos isolados. Esta direção da fronteira é central para a transparência da solução, pois permite que a aplicação continue a chamar funções, aceder a propriedades, instanciar classes e encadear métodos como se estivesse a usar a biblioteca original, mas o estado comportamental da biblioteca permanece no contexto isolado.

Esta opção resulta da comparação entre as alternativas disponibilizadas pela Arena e os requisitos de compatibilidade da solução. Uma conversão por `isMarshalable: "json"` é adequada para dados simples, mas perde funções, protótipos e comportamento. Uma conversão completa, como `isMarshalable: true`, preserva mais estrutura, mas continua a depender de objetos reconstruídos, funções ponte e nova conversão do `this` quando um método é chamado. O registo explícito de pares com `registeredObjects` é útil para objetos conhecidos à partida, mas não resolve, de forma genérica, instâncias criadas dinamicamente por bibliotecas dentro de QuickJS (Re:Earth 2026). Por esse motivo, a implementação escolhe uma Membrana VM-Host para objetos comportamentais, o que permite que a aplicação manipule referências mediadas, enquanto a operação real continua a executar sobre a instância original no ambiente isolado.

A função `bindSandboxExport` envolve a superfície exportada pela biblioteca. Quando a aplicação acede a uma propriedade que é uma função, a membrana cria uma função ligada ao caminho correspondente dentro do ambiente isolado. De forma a chamar a função real dentro do contexto QuickJS, a função invoca uma *closure* criada com `arena.evalCode`. Quando a aplicação acede a um objeto aninhado, a membrana aplica novamente `bindSandboxExport`. Esta lógica permite que exportações com métodos estáticos e objetos compostos continuem utilizáveis a partir da aplicação.

O tratamento de valores devolvidos é feito por `maybeWrapResult` e `wrapReturned`. A primeira função decide se o resultado deve atravessar como valor ou como referência mediada.

A segunda cria a referência mediada quando o resultado é um objeto com comportamento próprio. A implementação evita envolver tipos que devem ser convertidos ou preservados como valores, como `TypedArray`, `ArrayBuffer`, `Date`, `RegExp`, `Error`, arrays e objetos `Arguments` - algo descrito em maior detalhe na Subsecção 6.4.4. Para objetos com comportamento próprio, guarda a referência no mapa temporário dentro do ambiente isolado e devolve um *proxy* que chama métodos ou lê propriedades sobre o objeto isolado. O Excerto 6.9 mostra a chamada de métodos sobre esse objeto armazenado.

```
1 if (typeof value === "function") {  
2   if (!fnCache.has(key)) {  
3     fnCache.set(key, (...args) => {  
4       const result = sbHelpers.tempCall(id, key, ...args);  
5       return maybeWrapResult(arena, result, sbHelpers);  
6     });  
7   }  
8   return fnCache.get(key);  
9 }
```

Listing 6.9: Chamada de método sobre objeto mantido no contexto QuickJS

Esta decisão resolve um problema importante de compatibilidade, onde métodos de objetos com estado devem executar com o `this` original do ambiente isolado. No modelo genérico da Arena, uma função exportada para o ambiente hospedeiro é chamada através de um *wrapper* que volta a converter o `this` do hospedeiro e os argumentos para *handles* QuickJS. Se esse recetor no ambiente hospedeiro já não corresponder exatamente à instância criada dentro do ambiente isolado, a chamada pode ser executada sobre uma cópia estruturalmente semelhante, mas distinta em identidade. Esta diferença é irrelevante para muitos objetos simples, mas quebra objetos cujo comportamento depende da identidade inicializada pelo construtor, como classes com campos privados, estado associado por `WeakMap`, *internal slots* ou getters definidos no protótipo. A membrana evita este problema ao manter o objeto vivo em `__SANDBOX_TEMP__` e ao executar métodos e leituras de propriedades dentro do ambiente isolado, com o recetor original. Por isso, a implementação usa referências mediadas para objetos comportamentais e reserva o marshalling para valores que devem efetivamente atravessar a fronteira.

O caso de construtores recebe tratamento próprio. A função `createInstance`, exposta pelo Gestor da *sandbox*, cria a instância diretamente dentro do módulo isolado e guarda-a de imediato em `__SANDBOX_TEMP__`. Assim, a instância não passa por uma ida e volta intermédia que poderia alterar a identidade do objeto. Este detalhe foi necessário para preservar classes e objetos com estado em bibliotecas onde se notaram falhas na avaliação de compatibilidade descrita na Secção 7.1.

6.4.4 Marshallers Personalizados

Os marshallers personalizados concretizam o mecanismo de extensão estabelecido em RNF-M02. São usados para valores que devem atravessar a fronteira entre ambientes, mas cujo tratamento genérico da Arena não preserva a representação ou a semântica esperada no lado de destino. Esta preocupação é distinta da Membrana VM-Host. A membrana anterior mantém objetos comportamentais no contexto onde foram criados. Os marshallers, por sua vez, definem conversões por valor para tipos cuja representação deve existir no lado de destino. A documentação de `quickjs-emsripten-sync` descreve várias opções relevantes para este ponto (Re:Earth 2026). `isMarshalable` controla se um objeto pode atravessar a

fronteira e se deve ser convertido por JSON ou por marshalling completo. `isWrappable` e `isHandleWrappable` controlam se um objeto do ambiente hospedeiro ou um *handle* do ambiente isolado pode ser envolvido por *proxies* de sincronização. Por fim, `customMarshaller` e `customUnmarshaller` permitem introduzir conversões específicas para tipos particulares (Re:Earth 2026).

A implementação usa estas opções de forma seletiva. A opção `isMarshalable` é mantida permissiva para permitir que valores necessários às bibliotecas atravessem a fronteira, mas a decisão sobre envolvimento é restringida pelos predicados dos `marshallers`. Quando um tipo, para o qual o marshalling genérico disponibilizado pela biblioteca não é suficiente, é detetado, `isWrappable` ou `isHandleWrappable` impedem que a Arena crie um *proxy* genérico, permitindo que o valor chegue ao `customMarshaller` ou ao `customUnmarshaller`. Esta abordagem é mais adequada do que usar apenas `isMarshalable`: "json", porque JSON perderia funções, protótipos e classes. Também é mais adequada do que permitir envolvimento genérico para todos os objetos, porque certos objetos nativos dependem de *internal slots* ou de verificações de tipo que não reconhecem *proxies* como instâncias válidas.

A implementação usa um agregador que instancia os `marshallers` disponíveis e produz as opções usadas pela Arena. Cada `marshaller` expõe predicados para impedir envolvimento genérico, uma conversão host-VM e uma conversão VM-host. O Excerto 6.10 mostra a composição dos `marshallers` no agregador. Depois de criada a configuração, o agregador devolve a própria configuração para o Gestor de *Sandbox*, que a passa para a Arena durante a inicialização.

```

1  const marshallers = [
2    createTypedArrayMarshaller(vm),
3    createRegExpMarshaller(vm),
4    createArgumentsMarshaller(vm),
5    createErrorMarshaller(vm),
6  ];
7
8  const arenaOptions = {
9    isMarshalable: true,
10   isWrappable: (value) =>
11     marshallers.every((m) => m.isWrappable(value)),
12   isHandleWrappable: (handle, ctx) =>
13     marshallers.every((m) => m.isHandleWrappable(handle, ctx)),
14   customMarshaller: marshallers.map((m) => (value, ctx) =>
15     m.marshal(value, ctx)
16   ),
17   customUnmarshaller: marshallers.map((m) => (handle, ctx) =>
18     m.unmarshal(handle, ctx)
19   ),
20 };

```

Listing 6.10: Composição dos `marshallers` na configuração da Arena

Os tipos seleccionados são baseados nos exemplos apresentados no Capítulo 4 e demonstram diferentes utilizações do ponto de extensão. O `marshaller` de `TypedArray`, relevante para bibliotecas que manipulam dados binários, como `image-js` e `tweetnacl`, copia vistas (*views*) binárias entre ambiente hospedeiro e isolado. Esta cópia é necessária porque um `TypedArray` transformado num *proxy* genérico deixa de satisfazer verificações como `ArrayBuffer.isView` dentro do ambiente isolado e pode quebrar operações sobre buffers. O `marshaller` de `RegExp` preserva `source` e `flags`, evitando erros de recetor incompatível

em métodos nativos de expressões regulares. O marshaller de `Arguments` reconstrói a forma esperada por código que distingue estes objetos de objetos comuns, como acontece em operações da biblioteca `lodash`. O marshaller de `Error` preserva nome, mensagem, `stack` e tipo de erro quando possível, o que é relevante para bibliotecas de validação, como `zod` e `yup`, e para código que usa `instanceof` ou verificações de exceção.

Esta composição mantém o Gestor da *sandbox* independente dos tipos concretos suportados. Para adicionar suporte a um novo tipo, basta implementar um novo marshaller e registá-lo no agregador, mantendo estável a lógica principal de criação da *sandbox*. Os marshallers implementados demonstram a utilização deste mecanismo e aumentam a compatibilidade com os padrões considerados, mas não representam todos os tipos que podem exigir tratamento específico nem garantem compatibilidade universal com todas as bibliotecas ou valores JavaScript. Depois de estabelecida esta fronteira de execução transparente, a secção seguinte descreve uma extensão opcional para reduzir o número de travessias em fluxos de maior granularidade.

6.5 Implementação do Modo Coarse-Grained

O modo *coarse-grained* foi implementado como extensão opcional ao modo transparente. A sua responsabilidade é deslocar uma sequência de operações para dentro do mesmo ambiente isolado, expondo ao ambiente hospedeiro apenas uma função agregada. Desta forma, cumpre a sua finalidade de reduzir travessias repetidas da fronteira host-VM em caminhos de execução onde a aplicação não precisa de observar cada chamada intermédia. Esta secção concentra os detalhes de implementação dos adaptadores de carga de trabalho, mantendo-os separados do fluxo transparente descrito nas secções anteriores.

A configuração de cargas de trabalho acrescenta, dentro da entrada da biblioteca, um objeto `workloads`. Cada carga de trabalho declara o identificador do módulo adaptador e os nomes exportados que devem ficar disponíveis para a aplicação. O Excerto 6.11 mostra um exemplo com `lodash`.

```
1 export default {  
2   "lodash": {  
3     permissions: [],  
4     workloads: {  
5       collection: {  
6         entry: "collection-workload",  
7         exports: ["summarizeOrders"],  
8       },  
9     },  
10  },  
11 };
```

Listing 6.11: Exemplo de configuração com adaptador de carga de trabalho

Neste modo, o plugin executa trabalho adicional. Além do *proxy* transparente da biblioteca e do bundle isolado da dependência, gera um *proxy* específico para a carga de trabalho e um bundle separado para o adaptador. A importação da aplicação não aponta para a biblioteca original, mas para um identificador da forma `sandbox-workload/<library>/<workload>`. O plugin redireciona esse identificador para o *proxy* de carga de trabalho gerado.

O bundle da carga de trabalho é criado por outro *child compiler*. A diferença principal está no uso de `ExternalsPlugin`. O adaptador importa a biblioteca através de `sandbox-real/<library>`

e, durante a geração do bundle da carga de trabalho, esse identificador é tratado como externo e resolvido para `__SANDBOXED_LIB__`. Assim, quando a carga de trabalho é avaliada dentro do ambiente isolado, usa a biblioteca real já carregada no mesmo contexto QuickJS.

```
1 new ExternalsPlugin("var", {  
2   ['sandbox-real/${workload.libraryName}']: "__SANDBOXED_LIB__",  
3 }).apply(childCompiler);
```

Listing 6.12: Ligação da carga de trabalho à biblioteca já carregada na *sandbox*

O *proxy* transparente da biblioteca passa a incluir funções auxiliares para carregar bundles de carga de trabalho quando necessário. A função `__getSandboxWorkloadExport` verifica se a carga de trabalho já foi carregada. Se ainda não foi, lê o bundle correspondente, chama `librarySandbox.loadWorkload` e devolve a exportação pedida. Este carregamento reutiliza a *sandbox* da biblioteca, não cria uma segunda instância QuickJS nem uma segunda cópia da dependência.

O *proxy* específico da carga de trabalho é simples. Importa `__getSandboxWorkloadExport` a partir do *proxy* da biblioteca, obtém a função agregada e reexporta uma função com o mesmo nome para a aplicação. O resultado é que a aplicação passa a invocar uma operação de maior granularidade, sendo a sequência interna de chamadas executada dentro da *sandbox*.

```
1 const _workload_summarizeOrders =  
2   await __getSandboxWorkloadExport("collection", "summarizeOrders");  
3  
4 export function summarizeOrders(...args) {  
5   return _workload_summarizeOrders(...args);  
6 }
```

Listing 6.13: Forma simplificada de um *proxy* de carga de trabalho

O adaptador contém a lógica agregada que deve permanecer dentro da *sandbox*. No caso da `lodash`, a carga de trabalho importa a biblioteca por `sandbox-real/lodash` e executa operações de filtragem, agrupamento, ordenação e agregação antes de devolver apenas o resultado final. O Excerto 6.14 mostra uma versão reduzida desse adaptador.

```
1 import _ from "sandbox-real/lodash";
2
3 export function summarizeOrders(orders) {
4   const completedOrders = _.filter(orders, { status: "completed" });
5   const totalsByRegion = _.mapValues(
6     _.groupBy(completedOrders, "region"),
7     (regionOrders) => _.sumBy(regionOrders, "total"),
8   );
9
10  return {
11    totalsByRegion,
12    premiumCustomerNames: _.chain(completedOrders)
13      .filter((order) => order.customer.tier === "premium")
14      .orderBy(["total", "customer.name"], ["desc", "asc"])
15      .take(10)
16      .map("customer.name")
17      .value(),
18  };
19 }
```

Listing 6.14: Exemplo reduzido de adaptador de carga de trabalho para lodash

A diferença prática face ao modo transparente está na localização das chamadas intermediárias. No modo transparente, a aplicação conduz cada chamada através do *proxy* da biblioteca, cada operação pode atravessar a fronteira *host-VM*. No modo *coarse-grained*, a aplicação chama uma única função agregada, e as chamadas internas à biblioteca ocorrem no ambiente isolado. Esta opção reduz o número de travessias para fluxos específicos, mas exige que o desenvolvedor explicita o adaptador e altere o ponto de importação desse caminho de execução.

Esta extensão preserva o mesmo modelo de isolamento. A carga de trabalho executa no contexto QuickJS/Wasm da biblioteca correspondente, usa a mesma política de permissões e comunica com o ambiente hospedeiro através da mesma Membrana VM-Host. A diferença é a granularidade da API exposta à aplicação.

6.6 Síntese da Implementação

A Tabela 6.1 sintetiza a correspondência entre as decisões arquiteturais descritas no Capítulo 5 e os mecanismos implementados no protótipo. A tabela separa o comportamento transparente, que constitui o fluxo principal, da extensão *coarse-grained*, que é usada apenas para cargas de trabalho declaradas.

Tabela 6.1: Alinhamento entre conceção e implementação

Elemento de conceção	Implementação correspondente
Seleção de bibliotecas a isolar	Entradas por biblioteca na Configuração da <i>Sandbox</i>
Importação transparente	Redirecionamento com <code>NormalModuleReplacementPlugin</code> e <i>proxy</i> gerado
Geração automática de <i>proxies</i>	TypeScript Compiler API sobre <code>.d.ts</code> e <code>@types</code> , seguida de geração de reexportações
Separação entre aplicação e biblioteca isolada	Bundle principal importa o <i>proxy</i> ; <i>child compiler</i> produz o bundle isolado da biblioteca
Isolamento por biblioteca	Contexto QuickJS/Wasm dedicado criado pelo <code>SandboxManager</code>
Permissões granulares	<code>exposeWebAPIs</code> expõe apenas capacidades listadas na configuração
Membrana Host-VM	<code>bindWebApi</code> e <code>makeCallable</code> envolvem capacidades autorizadas do ambiente hospedeiro
Membrana VM-Host	<code>bindSandboxExport</code> , <code>wrapReturned</code> e <code>createInstance</code> preservam chamadas, estado e identidade
Marshalling extensível	<code>createArenaConfig</code> compõe <code>marshallers</code> para <code>TypedArray</code> , <code>RegExp</code> , <code>Arguments</code> e <code>Error</code>
Modo <i>coarse-grained</i>	Adaptadores de carga de trabalho configurados, bundle próprio, <i>proxy</i> de carga de trabalho e execução no mesmo contexto da biblioteca

Em síntese, a implementação mantém a separação definida na conceção: o processo de construção substitui importações e prepara artefactos, enquanto o *runtime* executa a biblioteca dentro de QuickJS/Wasm e medeia a comunicação através de permissões, membranas e `marshallers`. O modo transparente permanece o comportamento base. Os adaptadores de carga de trabalho acrescentam uma alternativa explícita para fluxos em que a granularidade das chamadas se torna parte extremamente relevante do custo de execução.

Capítulo 7

Avaliação

7.1 Avaliação de Compatibilidade

A avaliação de compatibilidade determina em que medida o comportamento validado pelas suites de testes das bibliotecas é preservado quando estas são executadas através da *sandbox*. A comparação usa a execução nativa como referência, permitindo separar falhas introduzidas pelo isolamento de falhas que já ocorrem no ambiente de avaliação. Os resultados caracterizam o protótipo nas bibliotecas e versões selecionadas, não constituem uma garantia de compatibilidade com todo o ecossistema npm.

7.1.1 Método Experimental

A seleção das bibliotecas considerou primeiro a sua relevância no ecossistema JavaScript. O State of JavaScript 2025 (Devographics 2025) e o State of Frontend 2024 (The Software House 2024) recolhem respostas da comunidade sobre as ferramentas e bibliotecas utilizadas pelos participantes. Estes levantamentos incluem Zod, Lodash, date-fns, Yup e Luxon, sustentando a avaliação de bibliotecas com utilização reportada por programadores (Devographics 2025; The Software House 2024).

O segundo critério foi a diversidade técnica. O conjunto abrange transformação de texto com strip-ansi, parsing com Marked, datas e internacionalização com date-fns e Luxon, validação de dados com Zod e Yup, operações sobre coleções com Lodash, criptografia com TweetNaCl, processamento de imagens com ImageJS e pesquisa de tipos MIME. Desta forma, os testes exercitam valores escalares, objetos aninhados, callbacks, objetos com estado, exceções, coleções nativas, TypedArray e ArrayBuffer. Lodash e strip-ansi permitem ainda relacionar esta avaliação funcional com os cenários de segurança apresentados na Secção 7.3.

Para cada biblioteca foi reutilizado um conjunto proveniente do respetivo projeto. A mesma versão da biblioteca, os mesmos dados de entrada e as mesmas asserções foram usados nos dois modos. No modo nativo, os testes importam diretamente a implementação da biblioteca. No modo isolado, as importações são redirecionadas para o *Proxy Gerado*, que conserva a interface pública e encaminha as operações para o contexto QuickJS.

O redirecionamento seguiu dois mecanismos, de acordo com o executor de testes. Os executores compatíveis com os mecanismos de carregamento do Node usaram redirecionadores próprios para módulos ES e CommonJS. Nos executores com resolução interna, como Jest e Vitest, o redirecionamento foi aplicado através de configurações, aliases ou plugins da própria framework. Estas adaptações pertencem ao mecanismo de teste, não modificam a biblioteca, os corpos dos testes nem as condições que determinam o seu sucesso.

Para cada modo foram contabilizados os testes executados e passados. O rácio de compatibilidade compara os testes passados na *sandbox* com os testes passados no modo nativo:

$$C = \frac{P_{\text{sandbox}}}{P_{\text{nativo}}} \times 100.$$

Esta definição desconta as falhas que já existem na execução de referência nativa. A atribuição de uma falha ao isolamento considera a identidade do teste e não apenas a diferença entre contagens agregadas. Quando a execução no modo isolado não completou todo o conjunto, o mesmo quociente é apresentado como limite inferior, pois os testes sem resultado não são classificados como sucessos. Para esses casos é apresentada também a taxa observada na parte executada, $P_{\text{sandbox}}/E_{\text{sandbox}}$, que descreve apenas os testes que chegaram a produzir um resultado.

7.1.2 Resultados de Compatibilidade

A Tabela 7.1 apresenta os resultados das dez bibliotecas. Os valores nativos mostram tanto a cobertura efetivamente executada como as falhas próprias do ambiente. A última coluna aplica o rácio definido anteriormente, sem atribuir à *sandbox* os testes que também falharam na execução de referência.

Tabela 7.1: Testes executados, testes passados e rácio de compatibilidade por biblioteca.

Biblioteca	Nativo executados	Nativo passados	Sandbox executados	Sandbox passados	Compatibilidade
date-fns ^a	3160	3081	2504	1888	$\geq 61,28\%$
ImageJS	40	40	40	37	92,50%
Lodash ^b	6734	6734	6734	5397	80,15%
Luxon	1196	1079	1196	957	88,69%
Marked ^a	165	165	159	88	$\geq 53,33\%$
MIME	15	13	15	12	92,31%
strip-ansi	8	8	8	8	100,00%
TweetNaCl	8914	8914	8914	8868	99,48%
Yup ^a	869	867	478	436	$\geq 50,29\%$
Zod	1982	1981	1982	1808	91,27%

^a A execução em *sandbox* não produziu resultados para todo o conjunto de testes. O rácio constitui um limite inferior face aos testes passados nativamente.

^b A biblioteca Lodash é contabilizada ao nível das asserções QUnit; as restantes linhas representam maioritariamente testes.

Strip-ansi completou todos os testes nos dois modos e preservou os oito resultados. TweetNaCl apresentou 8868 sucessos em 8914 testes, correspondendo a 99,48% da execução de referência. ImageJS, MIME e Zod mantiveram rácios entre 91,27% e 92,50%. No caso da biblioteca MIME, dois testes de linha de comandos falharam também nativamente no Windows e, por isso, não foram atribuídos à *sandbox*. A única falha nativa da biblioteca Zod foi igualmente excluída do denominador. A biblioteca Luxon registou 117 falhas nativas associadas sobretudo ao ambiente de internacionalização, enquanto a *sandbox* preservou 957 dos 1079 testes que passaram nativamente.

ImageJS falhou três testes que invocam `readSync`, uma API exclusiva de Node que não é exposta pela *sandbox* orientada ao browser. Os restantes testes, incluindo a transferência de dados binários, passaram. A biblioteca Lodash preservou 5397 das 6734 asserções. Este

resultado é apresentado separadamente das restantes bibliotecas porque a unidade reportada pelo QUnit é mais fina do que a contagem de testes usada pelos outros executores.

As três suítes incompletas exigem uma leitura adicional. Date-fns completou 247 dos 250 ficheiros e passou 1888 dos 2504 testes executados na *sandbox*, uma taxa observada de 75,40%. O limite inferior relativo à execução de referência completo é 61,28%. Yup completou 9 dos 14 ficheiros e passou 436 dos 478 testes executados, uma taxa observada de 91,21%. O limite inferior face à execução de referência é 50,29%. Na biblioteca Marked, 88 dos 159 testes registados passaram, correspondendo a 55,35% da parte executada e a um limite inferior de 53,33% face aos 165 sucessos nativos. Os testes complementares de conformidade da Marked obtiveram 637 sucessos em 652 casos CommonMark e 657 em 680 casos GFM, respetivamente 97,70% e 96,62%. A diferença mostra que a produção final de HTML é mais compatível do que os testes unitários que inspecionam ou alteram estruturas intermédias do lexer.

Adaptações do mecanismo de teste. As bibliotecas exigiram níveis diferentes de adaptação para que o mesmo conjunto de testes pudesse exercitar o *proxy*. Marked, MIME, strip-ansi e TweetNaCl usaram os redirecionadores de importações do Node, incluindo o suporte CommonJS necessário à TweetNaCl. Date-fns, ImageJS, Yup e Zod usaram configurações ou plugins de Vitest, enquanto a Luxon usou configurações próprias de Jest. Nesta última, o modo isolado requer módulos ES porque o *proxy* usa *top-level await*. Foi normalizada para sintaxe ES uma função auxiliar que combinava *require* e *exports*. A transformação preservou os corpos dos testes, os dados de entrada, as chamadas à biblioteca e as asserções, e o mesmo conjunto de testes adaptado foi executado nos dois modos.

7.1.3 Discussão das Causas de Incompatibilidade

As falhas adicionais não se distribuem de forma aleatória. Concentram-se em três propriedades que dependem da existência de um único *runtime* JavaScript ou de estado partilhado entre a aplicação e a biblioteca.

Identidade de referência e protótipos. Os valores QuickJS e os *wrappers* observados no ambiente hospedeiro pertencem a runtimes diferentes. A Membrana VM-Host preserva o recetor original para que métodos posteriores mantenham o *this*, os campos privados e os getters, mas não garante uma identidade canónica em todas as travessias. Por esse motivo, igualdade estrita, aliases de funções, protótipos e verificações com *instanceof* podem divergir. Este padrão surge nas comparações de Lodash e Yup, nos subtipos UTCDate e TZDate de date-fns, nas classes de Luxon e nos erros e schemas de Zod.

Aliasing e mutações partilhadas. Objetos, arrays e alguns valores especiais atravessam a fronteira por cópia ou reconstrução. A transferência preserva o valor observado num dado momento, mas não estabelece memória mutável partilhada. Uma alteração realizada por um callback do ambiente hospedeiro pode, assim, não ser refletida no objeto que permanece dentro da VM. Esta limitação explica grande parte das falhas unitárias da Marked, onde callbacks alteram arrays de tokens, e aparece também em metadados e opções de Yup, referências de JSON Schema da Zod e comparações da Lodash.

Tipos nativos e internal slots. Tipos como Set, Map, Date, RegExp, ArrayBuffer, DataView e objetos Intl possuem estado interno gerido pelo motor JavaScript. Um *proxy*

pode expor propriedades e métodos sem possuir os *internal slots* que esses métodos validam. A falha `Set object expected` da MIME, a perda dos subtipos de data em `date-fns` e os casos de coleções, `BigInt` e `RegExp` da Zod mostram os limites restantes. Os *marshallers* especializados reduzem estas diferenças para tipos suportados, mas não reproduzem toda a semântica.

Duas limitações secundárias completam esta explicação. A conversão de exceções pode preservar a rejeição e a mensagem sem manter o mesmo construtor no ambiente hospedeiro, como sucede nas 46 falhas da `TweetNaCl` e em erros ricos de `Zod` e `Yup`. Além disso, `date-fns` e `Yup` esgotaram o *heap* em ficheiros com criação intensiva de objetos. Este comportamento é consistente com a retenção de objetos `QuickJS` necessária para preservar receivers, embora os dados recolhidos não demonstrem que essa seja a única causa.

7.1.4 Síntese dos Resultados

Os resultados indicam maior compatibilidade em APIs que transformam dados e devolvem resultados consumidos por valor. `Strip-ansi`, `TweetNaCl`, `ImageJS` e `MIME` preservaram mais de 92% da respetiva execução de referência, apesar de exercitarem strings, dados binários, criptografia e uma coleção nativa. APIs dependentes de igualdade referencial, subclasses nativas, mutação bidirecional de estruturas intermédias ou criação intensiva de *wrappers* apresentaram mais divergências.

Este compromisso decorre da própria fronteira de isolamento: partilhar diretamente objetos e protótipos do ambiente hospedeiro eliminaria algumas diferenças, mas reduziria a separação que a solução procura impor. O protótipo preserva uma parte substancial do comportamento avaliado, mas aplicações que dependam das propriedades identificadas devem validar especificamente as APIs que utilizam. Como ameaça à validade, o conjunto de dez bibliotecas limita a generalização destes resultados ao restante ecossistema npm.

7.2 Avaliação de Desempenho no Browser

A avaliação de desempenho procura quantificar o custo introduzido pelo isolamento em aplicações Web executadas no browser. A comparação usa duas versões do mesmo cenário. No modo nativo, a aplicação importa e executa diretamente a biblioteca. No modo isolado, o Webpack redireciona a mesma importação para o *Proxy* Gerado e a biblioteca executa no contexto `QuickJS/WebAssembly`. Os dados de entrada, as operações realizadas e as validações funcionais permanecem iguais. Assim, a variável experimental é a introdução do mecanismo de isolamento.

7.2.1 Método Experimental

O mecanismo de teste concebido usa `Playwright` para controlar uma instância *headless* do Chromium (Microsoft 2026a) e o Chrome DevTools Protocol (CDP) para recolher métricas disponibilizadas pelo browser (Google 2026a). O Webpack produz, para cada cenário, os bundles nativo e isolado. A instrumentação acrescentada ao mecanismo de isolamento regista ainda os eventos internos, o número de valores que atravessa as membranas, a direção de cada travessia e uma estimativa do volume de dados transferido. A análise posterior é realizada em Python com recurso às bibliotecas `pandas` e `NumPy` para análise de dados e `Matplotlib` para visualização (Harris et al. 2020; Hunter 2007; McKinney 2010).

Os ensaios foram executados numa máquina com Windows 11, processador AMD Ryzen 7 4800H e 16 GB de memória, usando Node.js 24.13.0, Playwright 1.59.1 e Chromium 147.0.7727.15. Cada combinação entre cenário e modo incluiu duas iterações de aquecimento e dez iterações medidas. Em cada iteração, a função do cenário foi executada dez vezes. O processo Chromium foi reutilizado dentro da mesma combinação, mas cada iteração criou um novo contexto e uma nova página, reduzindo a reutilização de estado entre medições.

A ordem dos cenários foi determinada por uma permutação pseudoaleatória com uma seed fixa, permitindo a reprodutibilidade dos resultados. As execuções em modo nativo e isolado do mesmo cenário permaneceram adjacentes e a posição inicial dos dois modos foi alternada entre pares. Esta organização reduz a influência de variações temporais do sistema sobre a comparação.

Cada cenário representa uma utilização completa e verificável da biblioteca. O Excerto 7.1 apresenta uma versão reduzida da carga de trabalho transparente da biblioteca Lodash. A aplicação filtra encomendas, agrupa-as por região e calcula agregados. O mesmo módulo é compilado nos dois modos, pelo que não existe uma implementação simplificada específica para a *sandbox*.

```
1 import _ from "lodash";
2
3 export default function run() {
4   const completed = _.filter(orders, { status: "completed" });
5   const totalsByRegion = _.mapValues(
6     _.groupBy(completed, "region"),
7     (regionOrders) => _.sumBy(regionOrders, "total"),
8   );
9   const premiumNames = _.chain(completed)
10     .filter((order) => order.customer.tier === "premium")
11     .orderBy(["total", "customer.name"], ["desc", "asc"])
12     .take(10)
13     .map("customer.name")
14     .value();
15
16   validateResults(totalsByRegion, premiumNames);
17 }
```

Listing 7.1: Exemplo reduzido de carga de trabalho transparente da biblioteca Lodash

A principal métrica medida é `scenario_ms`, obtida pela divisão de `scenario_total_ms` pelas dez repetições. Esta métrica representa o tempo de uma utilização completa da biblioteca. A métrica `browser_elapsed_ms` cobre uma janela mais ampla, desde a preparação da iteração até ao seu encerramento, e inclui navegação, importação do cenário, inicialização da *sandbox*, recolha CDP e limpeza do contexto. Por esse motivo, não deve ser interpretada como sinónimo do tempo da carga de trabalho. O harness recolhe também o pico e o valor final do *heap* JavaScript através de `Performance.getMetrics`, usando a métrica `JSHeapUsedSize` reportada pelo CDP (Google 2026b).

As métricas internas distinguem a Membrana Host-VM, usada para expor capacidades do browser à biblioteca, da Membrana VM-Host, usada pela aplicação para interagir com a API exportada pela biblioteca. Um *crossing* corresponde a um valor transferido, e não necessariamente a uma chamada de função. O número de bytes é uma estimativa diagnóstica

calculada sobre a estrutura dos valores, com limites de profundidade e de número de nós. Não constitui, portanto, uma medição byte a byte do transporte realizado pelo *runtime*.

Para cada cenário foram calculadas as medianas das dez iterações medidas. O sobrecusto temporal corresponde à razão entre a mediana *sandbox* e a mediana nativa. O intervalo de confiança de 95% foi estimado através de 10 000 reamostragens bootstrap independentes dos dados observados para os modos nativo e *sandbox*, calculando a mediana em cada reamostragem, usando uma *seed* fixa. Esta opção reduz a dependência de pressupostos paramétricos sobre uma amostra pequena e assimétrica.

7.2.2 Resultados de Tempo de Execução

A Figura 7.1 apresenta o sobrecusto mediano das dez cargas de trabalho transparentes. Todos os cenários executaram corretamente nas duas configurações. No entanto, a magnitude do custo varia de forma acentuada entre bibliotecas.

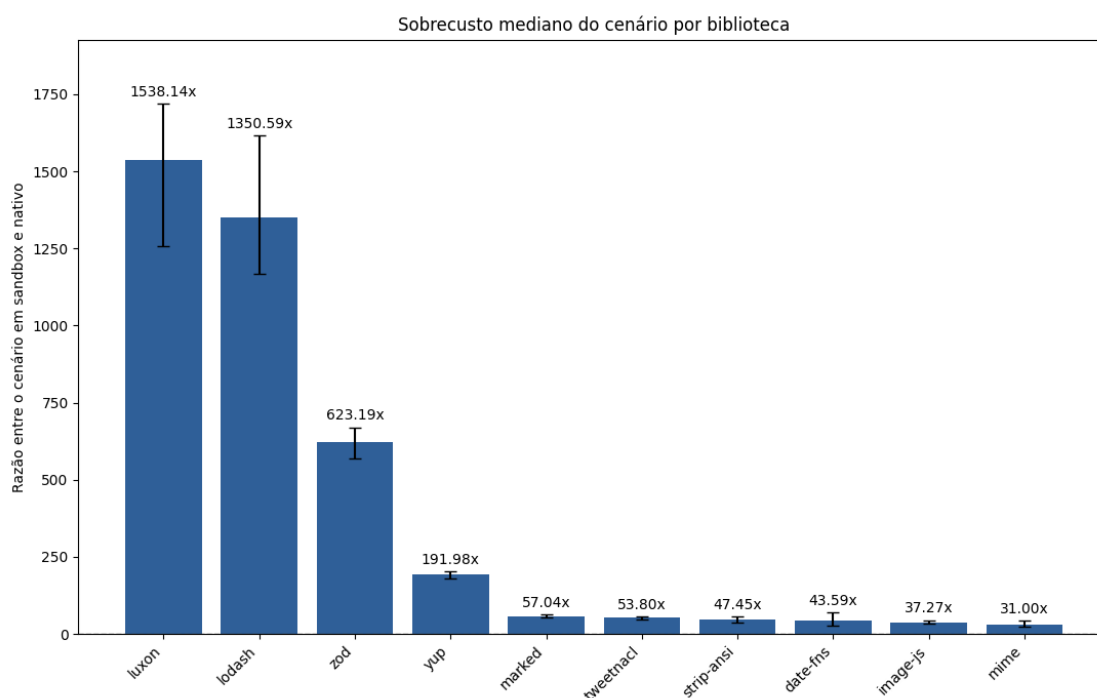


Figura 7.1: Razão entre o tempo mediano do cenário em *sandbox* e em execução nativa. As barras de erro representam o intervalo de confiança de 95% obtido por bootstrap.

A Tabela 7.2 complementa a razão relativa com os tempos absolutos. Esta distinção é necessária porque cargas de trabalho nativamente muito curtas podem produzir razões elevadas a partir de aumentos absolutos reduzidos.

Luxon, Lodash e Zod apresentam os maiores custos relativos. O cenário da biblioteca Luxon aumentou de 2,72 ms para 4176,04 ms por execução, correspondendo a 1538,14×. A biblioteca Lodash aumentou de 0,61 ms para 817,10 ms, ou 1350,59×. A Zod aumentou de 2,41 ms para 1505,00 ms, ou 623,19×. Estes valores mostram que o modo transparente pode ser insuficiente para cargas de trabalho que executam muitas operações mediadas sobre APIs compostas.

Tabela 7.2: Tempo mediano por execução do cenário e sobrecusto da *sandbox*.

Biblioteca	Nativo (ms)	Sandbox (ms)	Δ (ms)	Sandbox/Nativo	IC 95%
Luxon	2,72	4176,04	4173,32	1538,14×	[1258,92; 1720,27]
Lodash	0,61	817,10	816,50	1350,59×	[1169,29; 1616,34]
Zod	2,41	1505,00	1502,59	623,19×	[567,91; 670,10]
Yup	3,41	655,62	652,21	191,98×	[180,50; 203,47]
Marked	3,95	225,31	221,35	57,04×	[53,92; 62,06]
TweetNaCl	17,49	940,89	923,40	53,80×	[48,22; 56,56]
strip-ansi	0,10	4,74	4,64	47,45×	[36,50; 56,17]
date-fns	0,72	31,60	30,87	43,59×	[25,56; 70,95]
ImageJS	1,95	72,67	70,72	37,27×	[32,72; 45,22]
MIME	0,04	1,40	1,35	31,00×	[23,80; 42,75]

Nos restantes cenários, a razão relativa deve ser interpretada em conjunto com a diferença absoluta. Por exemplo, MIME e strip-ansi apresentam razões de 31,00× e 47,45×, mas os aumentos são apenas 1,35 ms e 4,64 ms. Date-fns e ImageJS acrescentam, respetivamente, 30,87 ms e 70,72 ms. Estes casos mantêm um custo absoluto inferior ao das cargas de trabalho que exigem comunicação ou execução mais intensiva.

O cenário de TweetNaCl constitui uma exceção útil para a interpretação. Apesar de registar apenas 11 travessias e 698 bytes estimados por execução, acrescenta 923,40 ms. A assinatura criptográfica executa cálculo intensivo dentro de QuickJS/WebAssembly. O resultado mostra que a diferença entre a execução JavaScript nativa do Chromium e a execução no motor QuickJS também pode dominar cargas de trabalho computacionais.

As Figuras 7.2 e 7.3 apresentam as distribuições obtidas no bootstrap. Para preservar a legibilidade, a maioria das bibliotecas é apresentada com uma escala comum até 100×, uma vez que os seus valores de sobrecusto se concentram numa gama semelhante. Luxon, Lodash, Zod e Yup são apresentados numa figura separada, com escala própria, por constituírem os quatro maiores *outliers* de sobrecusto de execução. Os intervalos não atravessam a razão unitária, pelo que a direção do efeito é consistente em todas as cargas de trabalho. Além disso, a mediana da distribuição das razões reamostradas pode diferir ligeiramente da razão calculada diretamente a partir das duas medianas observadas.

7.2.3 Influência das Membranas

As cargas de trabalho incluem padrões que exercitam intensamente as duas membranas. Lodash e Zod não usam APIs Web expostas nos cenários avaliados. O seu tráfego ocorre integralmente na Membrana VM-Host, quando a aplicação invoca a API da biblioteca e recebe valores ou objetos provenientes de QuickJS. O cenário da biblioteca Luxon, pelo contrário, depende de Intl para operações de calendários, fusos horários e formatação. A biblioteca chama repetidamente esta capacidade do ambiente hospedeiro, exercitando sobretudo a Membrana Host-VM.

A Tabela 7.3 apresenta os valores absolutos destas três bibliotecas. Luxon transfere menos bytes no total do que Lodash e Zod, mas efetua muito mais travessias. Das 2785 travessias por execução, 2586 pertencem à Membrana Host-VM. A maior parte dos 169343 bytes também atravessa essa membrana. Lodash e Zod, em contraste, não registam travessias no sentido VM-host.

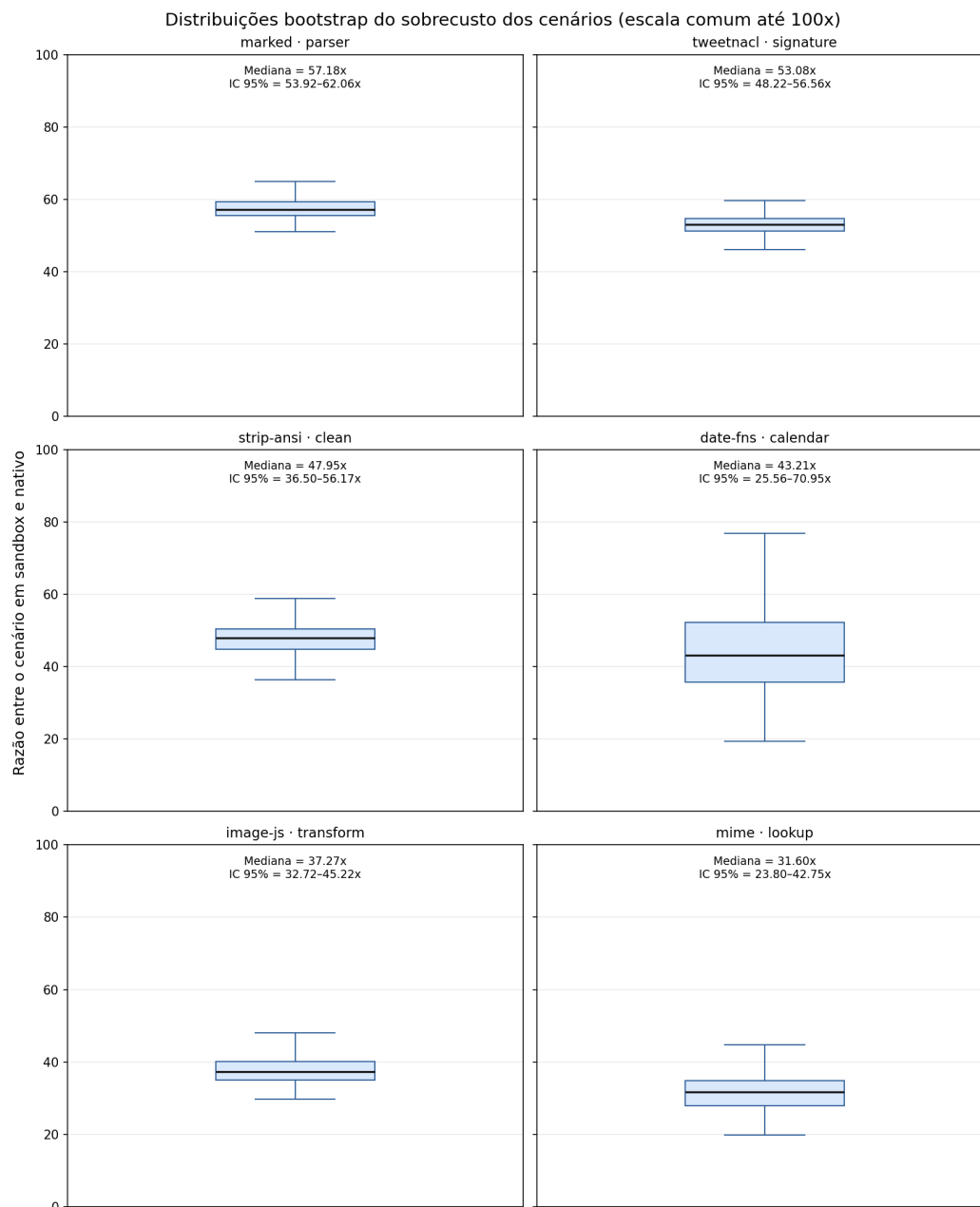


Figura 7.2: Distribuições *bootstrap* do sobrecusto de Marked, TweetNaCl, strip-ansi, date-fns, ImageJS e MIME, apresentadas numa escala comum até 100x.

Tabela 7.3: Diagnóstico das membranas nas cargas de trabalho com maior sobrecusto.

Biblioteca	Razão	Δ tempo (ms)	Travessias F/R	Bytes F/R	Bytes totais	Δ pico <i>heap</i> (MB)
Luxon	1538,14x	4173,32	2586 / 199	137271 / 32072	169343	93,92
Lodash	1350,59x	816,50	0 / 69	0 / 246308	246308	50,85
Zod	623,19x	1502,59	0 / 177	0 / 195875	195875	71,68

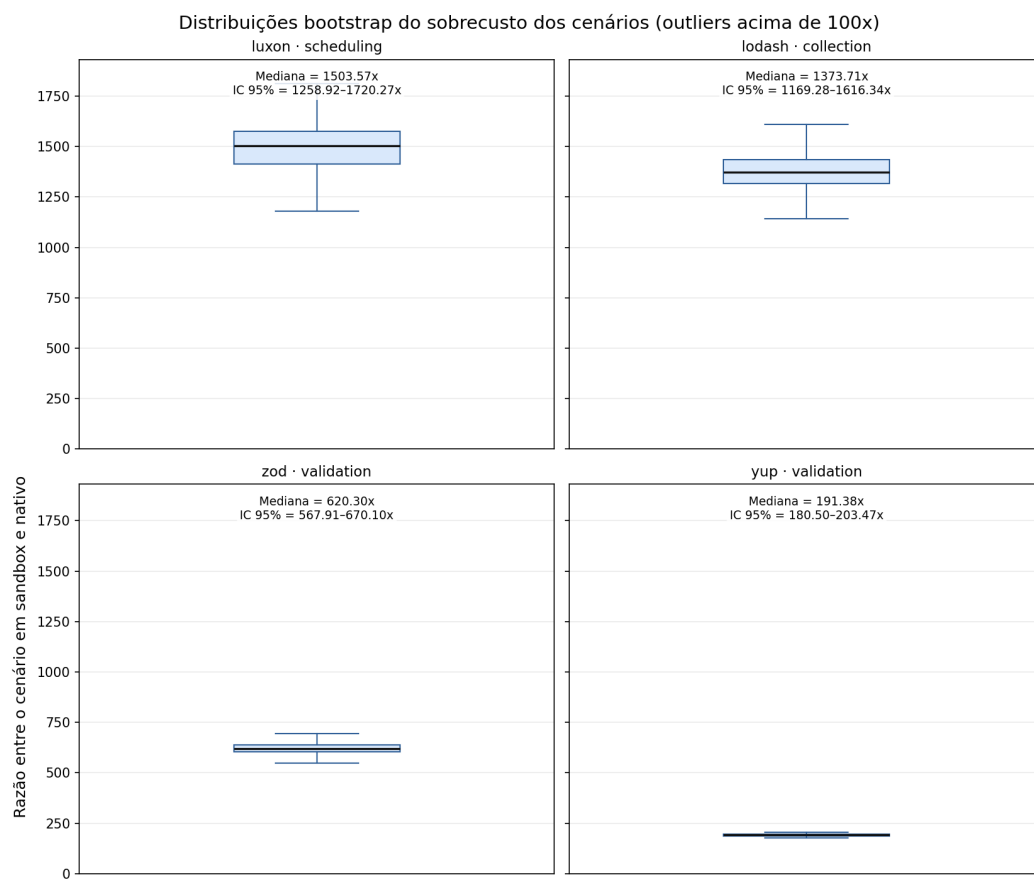


Figura 7.3: Distribuições *bootstrap* do sobrecusto de Luxon, Lodash, Zod e Yup, apresentadas numa escala própria por serem os quatro maiores *outliers* de sobrecusto de execução.

Este contraste ajuda a explicar por que razão o volume transferido, embora relevante, não determina sozinho o resultado. Luxon tem maior sobrecusto do que Zod apesar de transferir menos bytes, e tem também maior sobrecusto do que Lodash, que transfere aproximadamente 77 mil bytes adicionais. Uma hipótese compatível com os dados é que as chamadas muito frequentes a `Intl` acrescentem um custo fixo de mediação em cada travessia da Membrana Host-VM. A execução de métodos nativos expostos, a conversão dos argumentos e o retorno dos resultados repetem-se milhares de vezes. Os dados não isolam experimentalmente cada uma destas operações, pelo que esta interpretação deve ser entendida como explicação plausível, e não como demonstração causal.

A utilização de memória segue um padrão semelhante. Luxon apresenta o maior aumento do pico de *heap* JavaScript, 93,92 MB, seguido de Zod com 71,68 MB e Lodash com 50,85 MB. O isolamento introduz um custo fixo associado à inicialização de QuickJS, ao bundle e às estruturas da Arena. Para além desse custo, cargas de trabalho com mais valores e mais travessias podem manter representações temporárias, *proxies* e dados nos dois lados da fronteira. O pico de *heap* reportado pelo CDP corresponde à métrica `JSHeapUsedSize` devolvida por `Performance.getMetrics`, por isso, mede o *heap* JavaScript observado pelo Chromium e não toda a memória do processo ou a memória linear Wasm, sendo usado como indicador comparativo (Google 2026b).

7.2.4 Correlação com a Comunicação entre Contextos

As Figuras 7.4 e 7.5 mantêm as dez cargas de trabalho transparentes e relacionam o volume mediano transferido com o sobrecusto de tempo e de memória. O primeiro gráfico usa como variável dependente a razão temporal *sandbox*/nativo. O segundo usa a diferença absoluta do pico de *heap*. Em ambos se observa uma tendência crescente, embora Luxon, Lodash e Zod revelem que bibliotecas com volumes próximos podem apresentar custos distintos.

A Tabela 7.4 apresenta as correlações de Spearman mais relevantes. Para a memória, o analisador correlaciona as métricas internas com a razão entre os picos de *heap sandbox* e nativo, enquanto a Figura 7.5 mostra a diferença absoluta em MB. As duas representações são complementares: a diferença preserva a magnitude do consumo adicional e a razão permite comparar cargas de trabalho com linhas de base distintas.

Tabela 7.4: Correlações de Spearman entre comunicação nas membranas e sobrecusto.

Métrica explicativa	Correlação temporal	Correlação com pico de <i>heap</i>
Bytes totais	0,855	0,806
Número total de travessias	0,503	0,733
Bytes host-VM	0,891	0,842
Bytes VM-host	0,758	0,794
Bytes na Membrana VM-Host	0,818	0,770
Tamanho mediano de um crossing	0,317	-0,031

Entre as métricas agregadas, os bytes totais apresentam a associação mais forte com o sobrecusto temporal ($\rho = 0,855$) e com a razão do pico de *heap* ($\rho = 0,806$). A associação é ainda mais elevada para os bytes host-VM: 0,891 no tempo e 0,842 na memória. Estes valores são coerentes com o custo de copiar ou reconstruir dados que entram no contexto isolado. Os bytes na Membrana VM-Host apresentam também correlações elevadas, 0,818 e 0,770, o que é visível nas cargas de trabalho Lodash, Zod e Yup.

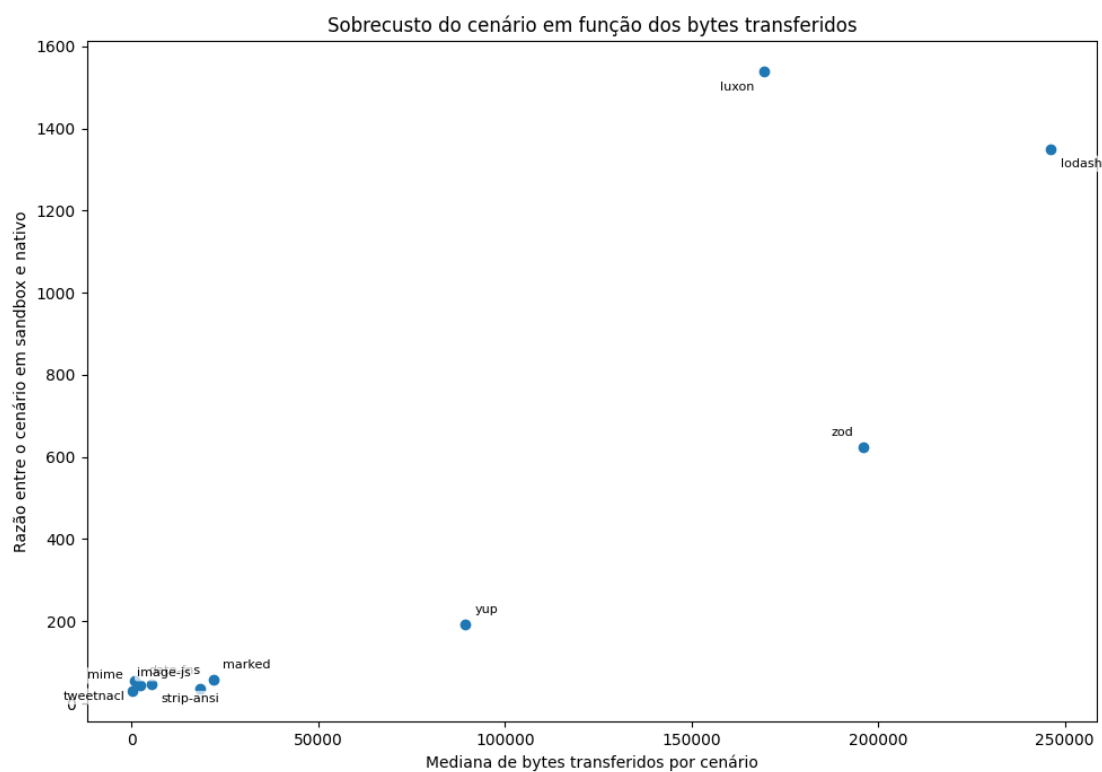


Figura 7.4: Sobrecusto temporal em função dos bytes estimados transferidos pelas membranas.

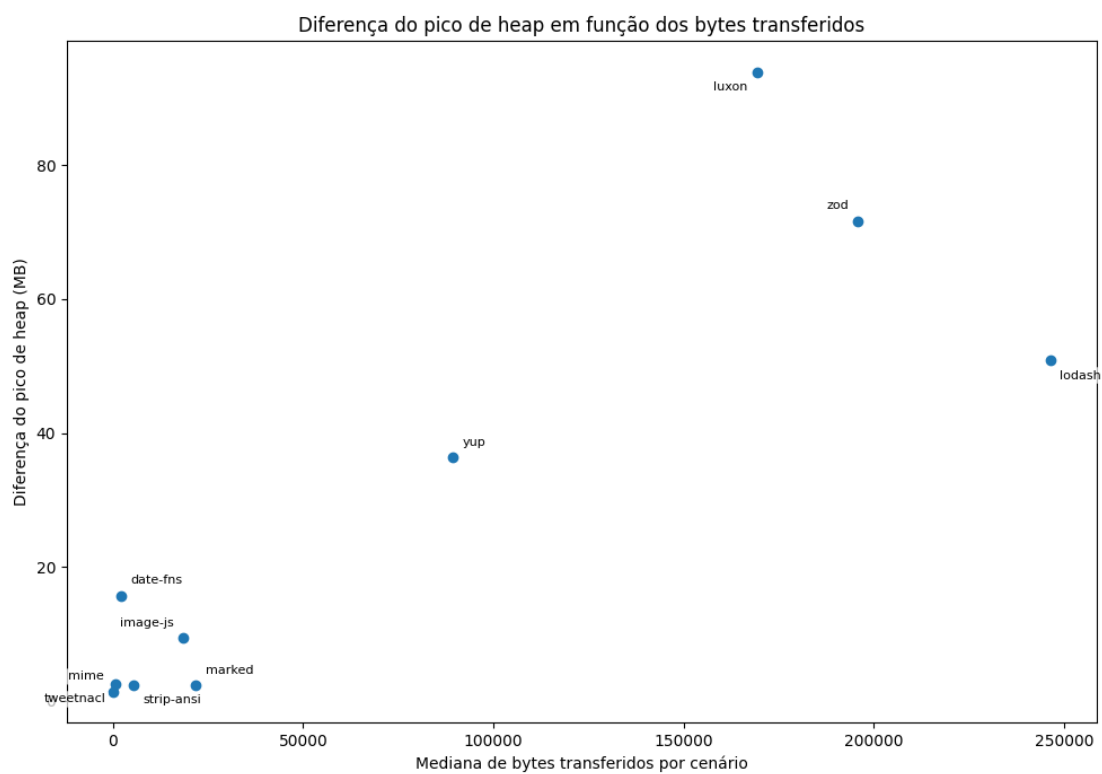


Figura 7.5: Diferença do pico de *heap* JavaScript em função dos bytes estimados transferidos pelas membranas.

O número de travessias apresenta uma correlação temporal moderada, 0,503, mas uma correlação mais forte com a memória, 0,733. Em sentido oposto, o tamanho mediano de uma travessia tem correlação fraca com o tempo, 0,317, e praticamente nula com a memória, -0,031. Em conjunto, estes resultados sugerem que o volume acumulado e a repetição da mediação explicam melhor o custo do que o tamanho de uma travessia típica. Luxon reforça esta leitura ao não liderar o volume total de bytes, mas combinando milhares de travessias com acesso reiterado a `Intl`.

As correlações descrevem apenas dez cargas de trabalho e não estabelecem causalidade. Além disso, as bibliotecas diferem na quantidade de computação executada dentro de QuickJS, na forma dos valores e no uso de APIs do ambiente hospedeiro. O caso TweetNaCl demonstra esse efeito, onde a carga de trabalho tem pouco tráfego de fronteira, mas mantém um custo temporal elevado devido ao cálculo criptográfico. A explicação mais consistente com o conjunto de resultados é, portanto, multifatorial: execução no motor QuickJS, frequência das travessias, volume acumulado transferido e utilização de capacidades do ambiente hospedeiro contribuem em proporções diferentes.

7.2.5 Adaptadores de Carga de Trabalho

O modo transparente privilegia a compatibilidade com a API original, mas permite que a aplicação conduza sequências longas de operações através da fronteira. Os adaptadores de carga de trabalho deslocam essa sequência para dentro do ambiente isolado e expõem ao ambiente hospedeiro apenas uma operação agregada. O Excerto 7.2 mostra esta diferença para a biblioteca Lodash: as operações intermédias são executadas no mesmo contexto da biblioteca e apenas o objeto de entrada e o resultado final atravessam a membrana.

```
1 import _ from "sandbox-real/lodash";
2
3 export function summarizeOrders(orders) {
4   const completed = _.filter(orders, { status: "completed" });
5   const totalsByRegion = _.mapValues(
6     _.groupBy(completed, "region"),
7     (regionOrders) => _.sumBy(regionOrders, "total"),
8   );
9
10  return {
11    totalsByRegion,
12    premiumNames: _.chain(completed)
13      .filter((order) => order.customer.tier === "premium")
14      .map("customer.name")
15      .value(),
16  };
17 }
```

Listing 7.2: Exemplo reduzido de adaptador de carga de trabalho da biblioteca Lodash

A Figura 7.6 compara o modo transparente e o adaptador nos cenários das bibliotecas Lodash e Zod. Na Lodash, a razão diminui de 1350,59× para 78,41×; na Zod, diminui de 623,19× para 16,21×. A melhoria coincide com a redução do tráfego na Membrana VM-Host: a biblioteca Lodash passa de 69 para 2 travessias e de 246308 para 9675 bytes; a Zod passa de 177 para 3 travessias e de 195875 para 10529 bytes.

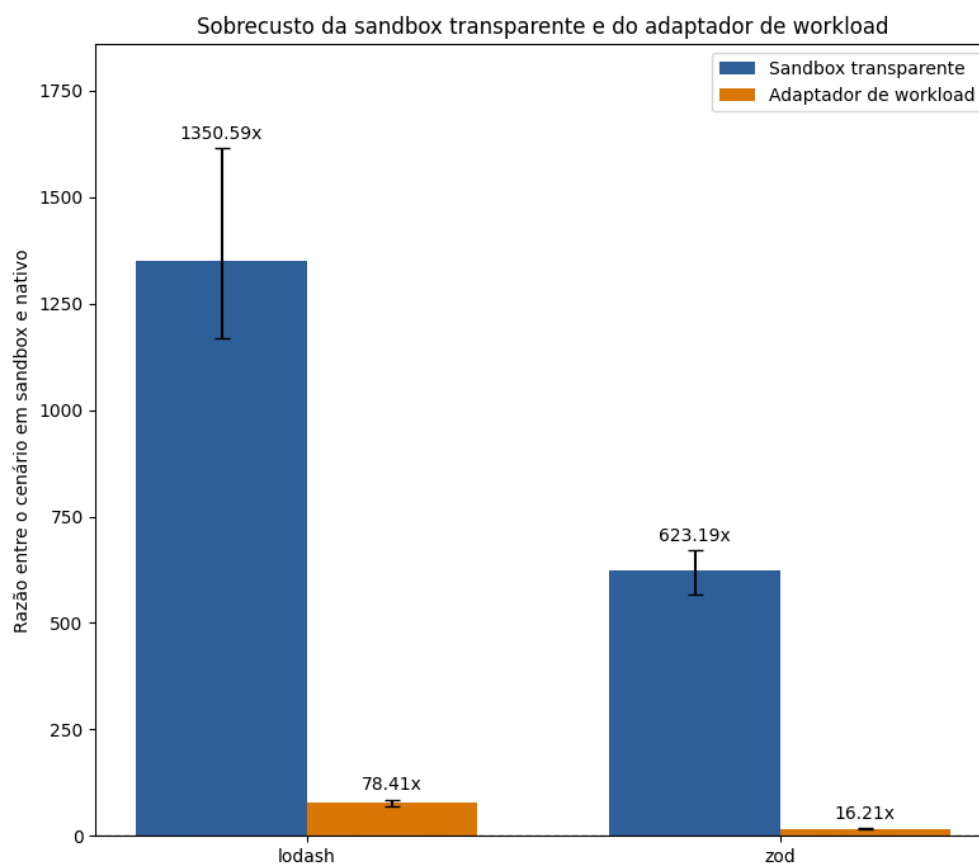


Figura 7.6: Comparação do sobrecusto do modo transparente e dos adaptadores de carga de trabalho de Lodash e Zod.

Esta otimização não oferece a mesma vantagem quando a própria biblioteca continua a chamar frequentemente uma capacidade do ambiente hospedeiro. No cenário da biblioteca Luxon, a agregação da lógica da aplicação não elimina as chamadas a `Intl`, pelo que a Membrana Host-VM permanece no caminho crítico. O benefício de um adaptador depende, assim, da possibilidade de substituir várias interações entre aplicação e biblioteca por uma operação local à *sandbox*. A contrapartida é exigir ao desenvolvedor uma função específica e um ponto de importação próprio para a carga de trabalho.

7.2.6 Síntese dos Resultados

Os resultados mostram que o sobrecusto não é uniforme entre bibliotecas. Quando a carga de trabalho exige poucas travessias, transfere pouco volume de dados e realiza computação moderada, o aumento absoluto mantém-se relativamente baixo, como acontece em MIME, strip-ansi, date-fns e ImageJS. O custo aumenta nos cenários que transferem estruturas maiores, executam muitas operações mediadas ou realizam computação intensiva em QuickJS.

O volume acumulado de bytes apresenta a correlação mais forte com o tempo e a memória, mas não explica isoladamente a ordenação das cargas de trabalho. A frequência das travessias e a direção da membrana ajudam a interpretar Luxon, enquanto o custo de execução dentro de QuickJS explica parte do comportamento de TweetNaCl. Para Lodash e Zod, os adaptadores de carga de trabalho demonstram que a redução da granularidade da comunicação diminui substancialmente o custo. Estes resultados satisfazem parcialmente RNF-P01: a solução torna o custo mensurável e mantém cenários pouco comunicativos em valores absolutos reduzidos, mas o modo transparente não oferece sobrecusto aceitável para todos os padrões de utilização.

7.3 Avaliação de Segurança

A avaliação de segurança verifica se os mecanismos implementados limitam o impacto de dependências adulteradas em três situações representativas: alteração de protótipos globais, intercepção de APIs sensíveis do browser e tentativa de acesso a uma capacidade não autorizada. O objetivo não consiste em executar código malicioso real, mas em reproduzir os seus efeitos relevantes de forma controlada e observar a diferença entre execução nativa e isolada.

7.3.1 Método Experimental

Os testes usam uma aplicação Web dedicada à avaliação, com manifesto, *lockfile* e instalação de dependências próprios. Os modos nativo e isolado executam o mesmo código da aplicação e a mesma instalação das dependências. A única diferença é a ativação do plugin de *sandbox*, que gera os bundles isolados, os *proxies* e redireciona as importações.

As modificações de strip-ansi e ImageJS são mantidas como patches locais através de patch-package. Estes patches não contêm o código original dos ataques, não efetuam pedidos externos e não acedem a recursos reais. O cenário Lodash usa diretamente uma versão vulnerável da biblioteca. Todos os testes usam canaries, endereços fictícios e implementações locais de `fetch`, `XMLHttpRequest` e uma carteira Web3. Esta opção reproduz os efeitos necessários para avaliar a mitigação sem contactar carteiras, serviços ou APIs externas, mantendo o ambiente experimental seguro e determinístico.

Cada cenário preserva também uma operação legítima da biblioteca. Esta condição verifica que o resultado de segurança não é obtido simplesmente impedindo toda a execução. As observações são comparadas com os valores esperados, apresentadas numa tabela no browser e convertidas num resultado PASS ou FAIL. Uma divergência provoca ainda uma exceção, tornando a validação reproduzível.

A Tabela 7.5 resume os três cenários, o comportamento legítimo, os efeitos observados na execução nativa e isolada, e os requisitos que cada cenário avalia.

Tabela 7.5: Síntese dos cenários de avaliação de segurança.

Cenário	Comportamento legítimo	Efeito nativo	Efeito <i>sandbox</i>	Requisitos
Lodash	Execução de <code>_.unset</code>	Eliminação da <i>canary</i> do protótipo global	<i>Canary</i> do ambiente hospedeiro preservada	RF-03, RNF-S02, RQ3
strip-ansi	Remoção de sequências ANSI	Interceção das três APIs e alteração dos destinatários	APIs e destinatários do ambiente hospedeiro inalterados	RF-03, RF-05, RNF-S01–S03
ImageJS	Transformação e codificação da imagem	Escrita adicional em <code>localStorage</code>	Operação legítima concluída e escrita bloqueada	RF-04, RF-05, RNF-S03

O cenário da biblioteca Lodash reproduz uma vulnerabilidade conhecida de poluição de protótipos onde um atacante pode usar a função `_.unset` para manipular protótipos globais, sendo esperado que o efeito seja confinado na *sandbox*. O cenário de strip-ansi simula a interceção de APIs sensíveis do browser, como `fetch`, `XMLHttpRequest` e `window.ethereum`, sendo esperado que a ausência de quaisquer capacidades configuradas não permita a execução das ações simuladas. O cenário de ImageJS tenta aceder a uma capacidade não autorizada, escrevendo em `localStorage`, sendo esperado que a operação seja bloqueada na *sandbox*, enquanto os acessos legítimos são permitidos.

7.3.2 Confinamento de Poluição de Prototype com Lodash

O primeiro cenário usa Lodash 4.17.21, uma versão publicada incluída no intervalo vulnerável do CVE-2025-13465, e reproduz o caminho canónico da vulnerabilidade. O advisory descreve a possibilidade de caminhos manipulados em `_.unset` e `_.omit` alcançarem protótipos globais e eliminarem propriedades herdadas, afetando a integridade ou a disponibilidade da aplicação (GitHub Advisory Database 2025). O cenário é também representativo da classe mais ampla de poluição de protótipos encontrada em pacotes npm (Houis et al. 2026).

Antes da chamada vulnerável, o teste introduz uma *canary* em `Object.prototype`. Em seguida, chama `_.unset` com o caminho `["constructor", "prototype", canaryKey]`, como mostra o Excerto 7.3. O valor de retorno e a *canary* são observados antes e depois da operação.

```

1 const canaryKey = "__security_canary__";
2 const canaryValue = "host-canary";
3
4 Object.prototype[canaryKey] = canaryValue;
5 const unsetReturned = _.unset(
6   {},
7   ["constructor", "prototype", canaryKey],
8 );
9 const hostCanaryAfter = Object.prototype[canaryKey];

```

Listing 7.3: *Canary* usada no cenário Lodash

Na execução nativa, `_.unset` devolveu `true` e eliminou a propriedade do protótipo da aplicação. Na execução isolada, a chamada também foi concluída, mas a *canary* do ambiente hospedeiro manteve o valor original. A Figura 7.7 apresenta as duas observações.

(a) Execução nativa

Lodash Prototype Pollution (native)				
(index)	observation	expected	observed	result
0	'Host canary before _.unset'	'host-canary'	'host-canary'	'PASS'
1	'Vulnerable _.unset return value'	'true'	'true'	'PASS'
2	'Host canary after _.unset'	'deleted'	'deleted'	'PASS'
▶ Array(3)				
Both modes execute the same application and lodash@4.17.21 installation.				
The vulnerable native dependency deleted the host prototype canary.				
Scenario result: PASS				

(b) Execução sandbox

Lodash Prototype Pollution (sandbox)				
(index)	observation	expected	observed	result
0	'Host canary before _.unset'	'host-canary'	'host-canary'	'PASS'
1	'Vulnerable _.unset return value'	'true'	'true'	'PASS'
2	'Host canary after _.unset'	'host-canary'	'host-canary'	'PASS'
▶ Array(3)				
Both modes execute the same application and lodash@4.17.21 installation.				
The import was redirected to the generated proxy; the host prototype canary remained intact.				
Scenario result: PASS				

Figura 7.7: Resultados do cenário de poluição de protótipos com Lodash.

A mitigação decorre da separação entre o estado JavaScript da biblioteca e o estado da aplicação. A biblioteca Lodash vulnerável executa dentro do contexto QuickJS, com memória isolada do módulo Wasm, e opera sobre os objetos intrínsecos desse contexto. Uma alteração a `Object.prototype` produzida durante essa execução não modifica diretamente o protótipo global do Chromium. O *proxy* permite invocar a função, mas não transforma os protótipos internos do QuickJS nos protótipos da aplicação. O resultado suporta RF-03, RNF-S02 e RQ3 para este vetor específico.

A solução mitiga o comportamento porque as APIs globais do browser não são herdadas pela biblioteca isolada. A configuração vazia impede que `fetch`, `XMLHttpRequest` e a carteira sejam inseridos no contexto QuickJS/Wasm. Consequentemente, o código adulterado não obtém as referências usadas pela aplicação e não as pode substituir. Este resultado relaciona-se com RF-03, RF-05 e RNF-S01–RNF-S03: o isolamento reduz a superfície acessível, a ausência de permissões bloqueia as capacidades e a aplicação mantém o seu estado.

7.3.4 Permissões Granulares com ImageJS

O terceiro cenário avalia se a solução preserva capacidades necessárias sem conceder acesso indiscriminado ao ambiente. O teste usa ImageJS 1.5.0 com um patch local no caminho de codificação de imagens. Depois de produzir o *Data URL*, o código adulterado tenta escrever uma *canary* em `localStorage`, conforme o Excerto 7.5.

```
1 const base64Data = new TextDecoder().decode(base64);
2 if (typeof localStorage !== "undefined") {
3   localStorage.setItem(
4     "__security_image_attack__",
5     "written-by-image-js",
6   );
7 }
```

Listing 7.5: Tentativa de acesso adicional introduzida em ImageJS

A política expõe `Intl`, `TextEncoder` e `TextDecoder`. Estas capacidades são necessárias para operações suportadas pela biblioteca e pelas suas dependências. No percurso avaliado, `TextDecoder` participa diretamente na construção do *Data URL*. A capacidade de acesso a `localStorage` não é declarada porque não faz parte das capacidades necessárias ao funcionamento da biblioteca.

Na execução nativa, a transformação para escala de cinzentos, o filtro e a codificação foram concluídos, mas o patch escreveu também a *canary* em `localStorage`. Na execução isolada, a imagem de 4 por 4 píxeis foi igualmente transformada e codificada, enquanto a escrita adicional permaneceu bloqueada. A *canary* já existente no armazenamento do ambiente hospedeiro também não foi alterada. A Figura 7.9 apresenta estes resultados.

(a) Execução nativa

image-js Granular Permissions (native)				
(index)	observation	expected	observed	result
0	'Image transformation'	'4x4 output'	'4x4'	'PASS'
1	'PNG Data URL encoding through TextDecoder'	'created'	'created'	'PASS'
2	'Patched image-js localStorage write'	'written-by-image-js'	'written-by-image-js'	'PASS'
3	'Existing host localStorage canary'	'host-storage-canary'	'host-storage-canary'	'PASS'
▶ Array(4)				
The installed image-js package contains a harmless evaluation patch in encodeDataURL that attempts to write a localStorage canary.				
Native execution completed the normal image operation and the additional localStorage write.				
Scenario result: PASS				

(b) Execução sandbox

image-js Granular Permissions (sandbox)				
(index)	observation	expected	observed	result
0	'Image transformation'	'4x4 output'	'4x4'	'PASS'
1	'PNG Data URL encoding through TextDecoder'	'created'	'created'	'PASS'
2	'Patched image-js localStorage write'	'blocked'	'blocked'	'PASS'
3	'Existing host localStorage canary'	'host-storage-canary'	'host-storage-canary'	'PASS'
▶ Array(4)				
The installed image-js package contains a harmless evaluation patch in encodeDataURL that attempts to write a localStorage canary.				
The configured TextDecoder capability allowed encoding, while the unconfigured localStorage capability remained unavailable.				
Scenario result: PASS				

Figura 7.9: Resultados do cenário de permissões granulares com ImageJS.

A mitigação resulta da seleção explícita das capacidades. O *runtime* resolve e expõe apenas os nomes presentes na política da biblioteca. ImageJS recebe os mecanismos de codificação necessários, mas não recebe `localStorage`, por isso, o ramo adulterado não consegue produzir o efeito adicional. O resultado demonstra RF-04, RF-05 e RNF-S03: a solução preserva a funcionalidade legítima e aplica negação por omissão às capacidades restantes.

7.3.5 Limitação na Transferência de Capacidades

As políticas anteriores controlam APIs globais disponibilizadas pelo ambiente hospedeiro. Contudo, é possível conceder capacidades a uma biblioteca através dos valores que a própria aplicação fornece como argumentos. Dados simples são normalmente copiados ou reconstruídos dentro do módulo isolado, pelo que a alteração de um objeto comum não equivale à sincronização direta do objeto original. Funções de callback, propriedades com getters ou setters e objetos com métodos podem, porém, transportar comportamento executável do ambiente hospedeiro. Uma dependência comprometida pode invocar esse comportamento mesmo que a capacidade correspondente não conste da lista de APIs globais.

Esta limitação não invalida os resultados dos três cenários, que não fornecem à biblioteca capacidades equivalentes às que se pretende bloquear. Implica, no entanto, que o desenvolvedor deve tratar callbacks e objetos comportamentais como concessões deliberadas de autoridade. A utilização mais restritiva deve privilegiar dados simples, callbacks com uma única responsabilidade e interfaces mínimas. Objetos abrangentes, como clientes de rede ou armazenamento, referências ao DOM, gestores de estado e serviços da aplicação, não

devem ser passados a uma dependência não confiável quando os respectivos métodos não são necessários.

Como trabalho futuro, a fronteira de argumentos pode ser reforçada sem remover o modo transparente atual. Uma primeira etapa consiste num modo de auditoria que identifique funções, propriedades de acesso (*accessors*) e instâncias com métodos. Políticas por biblioteca podem depois distinguir um modo compatível de um modo *data-only*. Para os casos que exigem callbacks, *wrappers* explícitos podem declarar os métodos autorizados, o número de chamadas e as regras aplicadas aos argumentos e resultados. Esta evolução torna a transferência de autoridade visível, mantendo a compatibilidade como opção para APIs JavaScript que dependem de valores comportamentais.

7.3.6 Síntese dos Resultados

Os três cenários exercitam propriedades complementares. Lodash verifica que a mutação de protótipos durante a execução isolada não afeta diretamente o protótipo da aplicação. *strip-ansi* demonstra que uma dependência adulterada não consegue intercetar APIs sensíveis que não lhe foram concedidas. ImageJS mostra que a política pode disponibilizar capacidades necessárias e, simultaneamente, bloquear uma operação adicional sobre *localStorage*.

Nos três casos, o comportamento legítimo foi preservado e o efeito de segurança observado na execução nativa foi mitigado na execução isolada. Os resultados sustentam a eficácia dos mecanismos de isolamento e de permissões para os vetores avaliados, em consonância com RQ3 e com os requisitos de segurança.

Capítulo 8

Conclusão

Este capítulo consolida o trabalho retomando o problema que o motivou: bibliotecas JavaScript/npm executadas no browser herdam, por omissão, a autoridade da aplicação que as integra. Quando uma dependência é comprometida, essa autoridade permite que o código adulterado aceda a capacidades sensíveis, modifique estado partilhado e produza efeitos fora da funcionalidade para a qual foi incluído. O trabalho investigou se uma arquitetura baseada em Wasm poderia reduzir este risco ao aplicar isolamento e permissões ao nível de cada biblioteca, sem eliminar a possibilidade de a aplicação continuar a usar dependências reais.

O capítulo começa por sintetizar o percurso realizado e responde depois à pergunta principal e às quatro subperguntas de investigação. De seguida, avalia o cumprimento dos objetivos e dos requisitos, consolida os contributos alcançados e explicita as limitações da solução e da avaliação. Por fim, apresenta linhas de trabalho futuro e encerra com a principal conclusão retirada do estudo.

8.1 Síntese do Trabalho Desenvolvido

A investigação iniciou-se pela caracterização da cadeia de abastecimento JavaScript/npm no contexto de aplicações Web. O risco não resulta apenas da publicação de um pacote vulnerável ou malicioso, mas também do modelo que o executa. A dependência tende a partilhar o contexto e as capacidades da aplicação, sem uma política nativa que limite a sua autoridade. A mutabilidade dos protótipos, as dependências transitivas e a dificuldade de observar código dinâmico ampliam este problema. Ficheiros de bloqueio, auditoria, autenticação e proveniência reduzem riscos noutras fases da cadeia, mas não restringem o comportamento durante a execução.

O estado da arte permitiu comparar diferentes unidades de isolamento. Processos e contentores são adequados para isolamento de aplicações no servidor, mas não constituem uma unidade prática para cada biblioteca no browser. Soluções centradas em JavaScript separam contextos ou interpõem acessos, mas dependem da correção da mediação e apresentam compromissos de granularidade, compatibilidade ou desempenho. O Wasm oferece uma fronteira de memória e importação explícita de capacidades. Como as bibliotecas JavaScript expõem interfaces dinâmicas, esta fronteira exigiu um motor incorporado e uma camada de interoperabilidade.

A partir desta análise foram definidos requisitos de isolamento por biblioteca, preservação da forma de utilização das dependências, negação de capacidades não autorizadas e configuração explícita de permissões. Os requisitos incluíram também compatibilidade, desempenho e extensibilidade. A arquitetura resultante separa o sistema em dois momentos. Durante

o processo de construção, um plugin Webpack identifica as bibliotecas configuradas, produz bundles próprios, gera *proxies* e redireciona as importações da aplicação. Durante a execução, cada biblioteca é avaliada num contexto QuickJS alojado num módulo Wasm. As capacidades autorizadas do ambiente hospedeiro e os valores exportados pela biblioteca atravessam a fronteira através de membranas com responsabilidades distintas.

O protótipo concretizou esta arquitetura. A membrana host-VM adapta capacidades autorizadas; a Membrana VM-Host mantém objetos comportamentais no módulo isolado e fornece referências mediadas à aplicação. Um ponto de extensão para *marshallers* trata tipos que necessitam de conversão própria, com implementações para `TypedArray`, `RegExp`, `Arguments` e `Error`. Além do modo transparente, o modo *coarse-grained* permite que adaptadores de carga de trabalho executem operações agregadas dentro da *sandbox* e reduzam travessias em caminhos muito comunicativos.

A avaliação analisou compatibilidade, desempenho no browser e segurança. A compatibilidade foi medida reutilizando suites de teste de dez bibliotecas e comparando cada execução em *sandbox* com a respetiva execução de referência nativa. A avaliação de desempenho executou dez cargas de trabalho no Chromium e relacionou o tempo medido com o volume, a frequência e a direção das travessias entre contextos. Por fim, três cenários de segurança avaliaram alteração de protótipos, interceção de APIs sensíveis e tentativa de acesso a uma capacidade não autorizada. Em conjunto, estes eixos permitiram observar não apenas se a biblioteca continuava funcional, mas também que proteção era obtida e qual o custo introduzido pela fronteira.

8.2 Resposta às Questões de Investigação

Pergunta principal. A pergunta principal procurou determinar em que medida uma arquitetura baseada em Wasm com isolamento granular poderia mitigar, no browser, o impacto dos vetores de ataque predominantes na cadeia de abastecimento JavaScript/npm, mantendo níveis aceitáveis de desempenho e usabilidade. Os resultados permitem responder afirmativamente no plano da viabilidade técnica. A arquitetura executou bibliotecas selecionadas num contexto separado, restringiu as capacidades globais disponíveis e mitigou os efeitos observados nos três cenários de segurança, sem impedir as operações legítimas exercitadas. A integração no processo de construção e os *proxies* gerados permitiram também manter a forma habitual de importação no modo transparente.

No entanto, observaram-se variações de desempenho e usabilidade entre bibliotecas. O custo depende do trabalho realizado dentro do módulo isolado, do volume de dados transferido, da frequência das travessias e da necessidade de chamar capacidades do ambiente hospedeiro. O modo transparente foi adequado a cenários com pouca comunicação em termos de custo absoluto, mas apresentou sobrecusto elevado em cargas de trabalho com APIs compostas ou muitas operações mediadas. Os adaptadores de carga de trabalho reduziram esse custo quando foi possível deslocar a sequência de chamadas para dentro da *sandbox*, à custa de uma integração explícita. Assim, a arquitetura demonstrou que a mitigação granular é possível, mas a manutenção de níveis aceitáveis de desempenho e usabilidade depende do perfil da biblioteca e do padrão de utilização.

RQ1: vetores de ataque e vulnerabilidades arquiteturais. Os vetores analisados convergem num problema de autoridade ambiental. Depois de integrada numa aplicação Web, uma dependência executa normalmente com acesso às mesmas APIs globais e ao mesmo

estado JavaScript que o código da aplicação. Um pacote comprometido pode, por isso, interceptar APIs do browser, exfiltrar ou alterar dados e modificar estruturas globais. A poluição de protótipos constitui uma manifestação específica desta partilha, onde uma mutação introduzida por uma biblioteca pode influenciar objetos e componentes que não participaram na operação vulnerável.

A dimensão transitiva do ecossistema e o carregamento dinâmico agravam este risco. O desenvolvedor pode não conhecer diretamente todas as dependências presentes na aplicação, e a análise estática não consegue antecipar todos os caminhos determinados durante a execução. No âmbito do browser estudado nesta dissertação, as vulnerabilidades arquiteturais mais relevantes são, portanto, a ausência de uma fronteira por dependência, a partilha de estado global e a inexistência de um modelo nativo de capacidades aplicado individualmente a cada biblioteca. O comprometimento da publicação introduz o código adulterado, mas é o modelo de execução que determina a autoridade que esse código consegue exercer.

RQ2: limitações das técnicas convencionais. As técnicas convencionais atuam em fases diferentes e continuam a ser necessárias, mas não resolvem isoladamente o problema em tempo de execução. Ficheiros de bloqueio tornam a resolução de dependências mais reproduzível, mas podem fixar uma versão comprometida. A auditoria depende de vulnerabilidades conhecidas. A autenticação, as assinaturas e a proveniência reforçam a integridade e a rastreabilidade da publicação, mas não demonstram que o comportamento de um pacote legítimo é seguro em todos os contextos. Apesar de reduzirem a probabilidade de determinados compromissos, estas medidas não limitam a autoridade de código em execução.

Processos, máquinas virtuais e contentores fornecem fronteiras mais amplas e são adequados ao isolamento de aplicações ou tarefas no servidor. A sua unidade de execução e dependência do sistema operativo não se ajustam à integração de cada biblioteca num bundle de browser. Modelos de permissões aplicados à aplicação, como os encontrados em alguns runtimes JavaScript, podem continuar a partilhar a permissão com todas as dependências. Por sua vez, *sandboxes* implementadas por interposição JavaScript e propostas como o *ShadowRealm* separam parte do estado ou da superfície global, mas mantêm limitações relativas ao *heap*, às capacidades, aos recursos ou à robustez da mediação. A lacuna identificada é, assim, uma fronteira leve, aplicável por biblioteca no lado do cliente e acompanhada por uma política explícita de capacidades.

RQ3: eficácia do WebAssembly para isolamento granular. O Wasm fornece uma base adequada para impor uma fronteira de memória e impedir que o código nela contido obtenha referências do ambiente hospedeiro sem uma exportação explícita. No protótipo, esta propriedade foi combinada com um contexto QuickJS próprio, uma política de negação por omissão e membranas que controlam a interoperabilidade. A eficácia observada resulta desta composição. O Wasm, isoladamente, não interpreta bibliotecas JavaScript, não gera *proxies* e não decide que APIs do browser podem ser usadas. Essas responsabilidades pertencem ao motor incorporado, ao plugin, à configuração e ao *runtime* desenvolvido.

Os três cenários de segurança exercitaram propriedades complementares. No caso da Lodash, a operação vulnerável modificou o protótipo no modo nativo, enquanto a *canary* do protótipo da aplicação foi preservada na *sandbox*. No cenário strip-ansi, o código adulterado interceptou `fetch`, `XMLHttpRequest` e a carteira fictícia em execução nativa; sem permissões, não obteve essas referências no contexto isolado. No cenário ImageJS, a política forneceu `Intl`, `TextEncoder` e `TextDecoder`, permitindo concluir a transformação da imagem, mas

não forneceu `localStorage`, bloqueando a escrita adicional. Em todos os casos, a operação legítima foi preservada e o efeito de segurança observado na execução de referência foi mitigado.

O controle não é absoluto. A política governa as capacidades globais expostas durante a criação da *sandbox*, mas a aplicação pode transferir autoridade nos próprios argumentos. Funções de callback, propriedades de acesso e objetos com métodos transportam comportamento executável do ambiente hospedeiro. Quando a aplicação fornece esses valores, a biblioteca pode invocá-los mesmo que a capacidade correspondente não esteja na lista de permissões globais. A resposta à RQ3 é, portanto, afirmativa para os vetores e para o modelo de capacidades avaliados, mas não constitui uma garantia contra autoridade explicitamente transferida pela aplicação nem contra todas as classes de ataque.

RQ4: comparação com outras técnicas de isolamento. A comparação realizada é arquitetural e bibliográfica. Face a abordagens baseadas em processos, a solução opera dentro do processo do browser e usa a biblioteca como unidade de isolamento. Face a mecanismos puramente JavaScript, acrescenta uma fronteira Wasm e mantém o estado da biblioteca num motor separado. A política por biblioteca fornece ainda uma granularidade que não existe quando todas as dependências herdaram permissões atribuídas à aplicação.

O desempenho observado impede, contudo, que resultados de Wasm próximo do nativo sejam generalizados para esta arquitetura. O protótipo executa JavaScript através de QuickJS e medeia a comunicação através de *proxies*, membranas e conversões. Em APIs muito comunicativas, estes elementos dominam o custo. Os adaptadores demonstraram que a granularidade da comunicação é uma variável de concepção: na biblioteca Lodash, a razão entre os modos *sandbox* e nativo diminuiu de $1350,59\times$ para $78,41\times$; na Zod, diminuiu de $623,19\times$ para $16,21\times$. A melhoria foi substancial, mas não eliminou o sobrecusto, e não se aplica da mesma forma quando a biblioteca continua a chamar frequentemente uma capacidade do ambiente hospedeiro.

Quanto à experiência de desenvolvimento, o modo transparente mantém as importações da aplicação e centraliza a seleção de bibliotecas e permissões numa configuração explícita. A compatibilidade com bibliotecas amplamente utilizadas por programadores JavaScript é alta, com sete das dez bibliotecas avaliadas a preservar mais de 80% da respetiva execução de referência, enquanto as restantes expuseram limitações associadas a identidade, protótipos, tipos nativos ou criação intensiva de objetos. Alguns executores de testes exigiram também mecanismos próprios de redirecionamento. Os adaptadores de carga de trabalho introduzem uma alternativa de desempenho, mas exigem que o desenvolvedor defina e importe uma operação agregada. Em comparação com outras técnicas, a proposta oferece isolamento e permissões ao nível pretendido, aceitando a troca de parte da transparência semântica e do desempenho por essa fronteira. A usabilidade depende, contudo, do perfil da biblioteca e do padrão de utilização, sendo alta no modo transparente e mais exigente quando é necessária a adaptação de cenários mais comunicativos.

8.3 Cumprimento dos Objetivos e Requisitos

Os seis objetivos específicos formulados na Secção 1.3 foram abordados. Os capítulos de Contexto e o Estado da Arte caracterizaram os riscos, limitações atuais e analisaram o Wasm. A Análise traduziu o problema em requisitos. A Concepção definiu a arquitetura de isolamento granular por biblioteca e controle de capacidades, tratando a Implementação de

a concretizar num protótipo. Por último, a Avaliação mediu compatibilidade, desempenho e eficácia de isolamento em cenários representativos.

Neste contexto, isolamento, configuração e extensibilidade têm correspondência direta em mecanismos implementados. Já a compatibilidade, desempenho e usabilidade dependem do comportamento observado e do alcance experimental. A Tabela 8.1 consolida esta distinção.

Tabela 8.1: Síntese do cumprimento dos requisitos da solução

Requisitos	Estado	Evidência principal
RF-01, RST-03	Cumprido	A configuração seleciona bibliotecas individualmente, e o plugin produz uma <i>sandbox</i> e artefactos próprios para cada dependência configurada.
RF-02, RNF-U01	Parcialmente cumprido	O modo transparente preserva as importações e grande parte da forma de uso, mas os adaptadores exigem um ponto de integração explícito.
RF-03, RNF-S02, RST-02	Cumprido	Cada biblioteca executa num contexto QuickJS/Wasm próprio. O cenário da biblioteca Lodash confirmou que a mutação interna não alterou diretamente o protótipo do ambiente hospedeiro.
RF-04, RF-05, RNF-S01, RNF-S03	Cumprido	A política expõe apenas capacidades declaradas; os cenários strip-ansi e ImageJS confirmaram negação por omissão e concessão seletiva.
RNF-U02	Implementado sem validação empírica direta	A configuração explicita bibliotecas e permissões, mas a sua compreensibilidade não foi avaliada através de um estudo com programadores.
RNF-P01	Parcialmente cumprido	O custo absoluto foi reduzido em cargas de trabalho com pouca comunicação e os adaptadores diminuíram o sobrecusto, mas o modo transparente não apresentou custo aceitável em todos os cenários.
RNF-M01	Cumprido no âmbito avaliado	A arquitetura e os pontos de extensão suportaram dez bibliotecas com padrões de API distintos, sem demonstrar compatibilidade com o ecossistema completo.
RNF-M02	Cumprido no âmbito avaliado	O agregador permite acrescentar marshallers sem alterar o Gestor da <i>Sandbox</i> , e quatro tipos receberam implementações específicas.
RST-01	Cumprido	A solução foi concebida para aplicações Web e a avaliação principal executou cargas de trabalho e cenários de segurança no Chromium.

Os estados “cumprido no âmbito avaliado” assinalam que existe implementação e evidência experimental, mas que o resultado não pode ser generalizado para todas as bibliotecas ou ataques. “Parcialmente cumprido” identifica requisitos cujo mecanismo existe, mas cujo resultado não foi uniforme. Esta leitura é particularmente importante para o RNF-P01, onde o trabalho quantificou o custo e demonstrou uma forma de o reduzir, mas não confirmou sobrecusto aceitável para todos os padrões de utilização.

8.4 Principais Contributos

O principal contributo do trabalho é a conceção e concretização de uma arquitetura de isolamento granular para bibliotecas JavaScript executadas no browser. A arquitetura combina uma fronteira QuickJS/Wasm por biblioteca com integração no processo de construção. Esta opção permite deslocar a execução da dependência para um contexto separado sem obrigar a aplicação a invocar diretamente uma API genérica de isolamento em cada utilização.

Um segundo contributo é o modelo de integração e de controlo de capacidades. O plugin Webpack gera *proxies* a partir da superfície declarada pelas bibliotecas, redireciona importações e prepara bundles para avaliação dentro de QuickJS. Em tempo de execução, a configuração associa permissões a cada biblioteca e aplica negação por omissão. A Membrana Host-VM adapta apenas as capacidades autorizadas, enquanto a Membrana VM-Host conserva objetos comportamentais no contexto onde foram criados. O mecanismo extensível de marshalling complementa estas membranas para valores que necessitam de representação própria nos dois contextos.

O protótipo inclui ainda duas granularidades de integração. O modo transparente privilegia a preservação das importações e da API original. O modo *coarse-grained*, concretizado por adaptadores de carga de trabalho, permite deslocar uma operação aplicacional completa para a *sandbox* e reduzir travessias. Esta segunda modalidade foi proposta, implementada e comparada em cargas de trabalho de Lodash e Zod. O resultado evidencia que a granularidade da API exposta pelo isolamento influencia diretamente o custo observado.

Outro contributo reside nos mecanismos de avaliação e na evidência recolhida. As suites de compatibilidade exercitaram dez bibliotecas com padrões distintos. O mecanismo de avaliação do desempenho no browser comparou cargas de trabalho equivalentes e registou tempos, memória e comunicação nas duas membranas. A aplicação de segurança reproduziu três efeitos controlados de dependências vulneráveis ou adulteradas, preservando simultaneamente uma operação legítima.

Os resultados caracterizam também os perfis mais adequados. APIs que recebem dados, realizam trabalho local e devolvem resultados por valor mostraram maior compatibilidade e custos absolutos mais contidos. Identidade, mutação bidirecional e operações mediadas frequentes expuseram os limites da fronteira. A viabilidade depende, portanto, da biblioteca e da forma de comunicação. O protótipo é uma contribuição exploratória e extensível, não uma solução universal ou pronta para adoção indiscriminada em produção.

8.5 Limitações Identificadas

Arquitetura e compatibilidade. O principal limite arquitetural é a diferença entre os dois runtimes JavaScript. Valores que atravessam a fronteira não passam a pertencer ao mesmo *realm*. A membrana preserva o recetor original em vários padrões de objetos com estado, mas não garante uma identidade canónica em todas as travessias. Por isso, igualdade estrita, cadeias de protótipos e verificações com `instanceof` podem divergir. Objetos transferidos por cópia ou reconstrução também não estabelecem memória mutável partilhada, o que limita callbacks que esperam alterar uma estrutura mantida no outro contexto.

Tipos nativos constituem outra fonte de incompatibilidade. Objetos como `Set`, `Map`, `Date`, `RegExp`, `ArrayBuffer` e `DataView` dependem de *internal slots* do motor. Marshallers especializados cobrem parte destes casos, mas não reproduzem toda a semântica. A conversão de exceções pode preservar mensagem e rejeição sem manter o construtor esperado no ambiente hospedeiro. A retenção de referências necessária para conservar objetos comportamentais contribui ainda para pressão de memória.

A integração está acoplada ao Webpack e a declarações TypeScript adequadas. Outros bundlers exigem substituição de módulos, geração de *proxies* e produção de bundles equivalentes. Formatos de módulo e executores de testes podem ainda necessitar de redirecionamento específico. A transparência da importação não elimina, portanto, todo o trabalho de integração.

Segurança. A política implementada controla capacidades globais que o *runtime* expõe à biblioteca. Não controla automaticamente toda a autoridade que a aplicação transfere através de argumentos. Callbacks, getters, setters e objetos com métodos podem funcionar como capacidades do ambiente hospedeiro. Esta limitação exige que o desenvolvedor trate valores comportamentais como concessões deliberadas de autoridade e evite fornecer interfaces mais amplas do que a operação necessita.

A eficácia depende da configuração, das membranas, dos marshallers, do QuickJS e do *runtime* WebAssembly. Permissões demasiado amplas aumentam a superfície disponível, e falhas de mediação podem introduzir caminhos imprevistos.

Validade experimental. A compatibilidade cobre dez bibliotecas e versões específicas, uma fração reduzida do npm. Três suites não produziram resultados para todos os ficheiros, pelo que os rácios são limites inferiores.

Os resultados de desempenho foram recolhidos numa máquina Windows 11 e numa versão específica do Chromium. Cada cenário teve dez iterações medidas, suficientes para caracterizar o ambiente experimental, mas não para generalizar para outros browsers, sistemas operativos ou dispositivos. As correlações usam dez cargas de trabalho e descrevem associações, não relações causais. Os bytes transferidos são estimativas estruturais. O pico de *heap* recolhido pelo Chrome DevTools Protocol representa o *heap* JavaScript observado pelo browser, não toda a memória do processo nem a memória linear WebAssembly. As conclusões relativas a desempenho devem, por isso, ser entendidas no contexto destes cenários e métricas.

Os três testes de segurança reproduzem efeitos controlados e determinísticos. Esta opção permitiu comparar execução nativa e *sandbox* sem contactar serviços externos nem executar cargas maliciosas perigosas, mas não cobre a diversidade de técnicas presente em incidentes reais. A avaliação também não comparou experimentalmente o protótipo com V8 Isolates, Cage4Deno, JSand ou *ShadowRealm*. A resposta à RQ4 combina a comparação bibliográfica com os resultados próprios da solução. Por fim, a usabilidade foi inferida a partir da preservação das importações, da configuração declarativa e do esforço exigido pelos adaptadores. Não foi realizado um estudo com programadores que medisse compreensão, tempo de integração ou probabilidade de erros de configuração.

8.6 Trabalho Futuro

As linhas de evolução seguintes decorrem diretamente dos limites observados e são apresentadas por ordem de impacto esperado sobre a aplicabilidade da solução.

1. **Generalizar a redução de travessias.** Os adaptadores de carga de trabalho demonstraram que deslocar uma sequência de operações para dentro da *sandbox* pode reduzir substancialmente o custo. O passo seguinte consiste em apoiar a identificação de *hot paths*, sugerir operações que beneficiem de agregação e reduzir o trabalho manual necessário para criar os adaptadores. A otimização deve distinguir tráfego VM–host de chamadas repetidas a capacidades do ambiente hospedeiro, uma vez que a agregação da lógica aplicacional não elimina necessariamente o segundo caso.
2. **Melhorar a gestão do ciclo de vida e da memória.** A preservação de objetos comportamentais requer referências a valores mantidos no ambiente isolado. É necessário introduzir uma política explícita de libertação, investigar a utilização de mecanismos como `FinalizationRegistry` e reduzir a retenção de objetos temporários. A avaliação futura deve separar a memória do ambiente hospedeiro, a memória linear `WebAssembly` e as estruturas mantidas pelo *runtime*, permitindo relacionar libertação de referências com os casos de esgotamento do *heap*.
3. **Expandir a preservação semântica.** Novos `marshallers` podem cobrir coleções, datas, `DataView`, objetos de internacionalização e outros tipos com *internal slots*. Além da conversão por tipo, permanece necessário melhorar a identidade canónica, a propagação de mutações e a reconstrução de exceções. APIs assíncronas complexas e `callbacks` bidirecionais devem ser exercitados de forma específica, pois combinam ciclo de vida, identidade e execução de tarefas pendentes.

8.7 Considerações Finais

Esta dissertação demonstrou a viabilidade técnica de isolar bibliotecas npm selecionadas num contexto QuickJS/Wasm dentro do browser, integrando a fronteira no processo de construção e controlando explicitamente as capacidades globais disponibilizadas. O protótipo preservou utilização funcional em padrões diversos e mitigou os três efeitos de segurança avaliados. Ao mesmo tempo, a avaliação tornou visíveis os custos dessa separação: diferenças semânticas entre runtimes, sobrecusto de execução e comunicação, pressão de memória e trabalho adicional quando o modo transparente não é adequado.

A principal conclusão não é que o Wasm elimina o risco da cadeia de abastecimento, mas que permite alterar o modelo de confiança aplicado às dependências. Em vez de conceder implicitamente a autoridade da aplicação a todo o código importado, a arquitetura introduz uma fronteira por biblioteca e exige que capacidades externas sejam declaradas. Esta mudança reduz a propagação do impacto de uma dependência comprometida nos cenários estudados. A sua aplicação prática requer, contudo, seleção consciente das bibliotecas, políticas mínimas e padrões de comunicação compatíveis com o custo da fronteira. O isolamento granular constitui, assim, uma camada de defesa adicional e mensurável para aplicações Web, cuja evolução depende de equilibrar segurança, compatibilidade e desempenho sem voltar ao modelo de confiança implícita que motivou o trabalho.

Bibliografia

- Abbadini, Marco et al. (jul. de 2023). «Cage4Deno: A Fine-Grained Sandbox for Deno Subprocesses». en. Em: *Proceedings of the ACM Asia Conference on Computer and Communications Security*. Melbourne VIC Australia: ACM, pp. 149–162. isbn: 979-8-4007-0098-9. doi: 10.1145/3579856.3595799. url: <https://dl.acm.org/doi/10.1145/3579856.3595799> (acedido em 16/12/2025).
- Agten, Pieter et al. (dez. de 2012). «JSand: complete client-side sandboxing of third-party JavaScript without browser modifications». en. Em: *Proceedings of the 28th Annual Computer Security Applications Conference*. Orlando Florida USA: ACM, pp. 1–10. isbn: 978-1-4503-1312-4. doi: 10.1145/2420950.2420952. url: <https://dl.acm.org/doi/10.1145/2420950.2420952> (acedido em 17/11/2025).
- Akbary, Syrus (20 de mai. de 2026). *Edge.js: Running Node apps inside a WebAssembly Sandbox · Blog · Wasmer*. url: <https://wasmer.io/posts/edgejs-safe-nodejs-using-wasm-sandbox> (acedido em 29/06/2026).
- AlHamdan, Abdullah e Cristian-Alexandru Staicu (2025). «Welcome to Jurassic Park: A Comprehensive Study of Security Risks in Deno and its Ecosystem». en. Em: *Proceedings 2025 Network and Distributed System Security Symposium*. San Diego, CA, USA: Internet Society. isbn: 979-8-9894372-8-3. doi: 10.14722/ndss.2025.230284. url: <https://www.ndss-symposium.org/wp-content/uploads/2025-284-paper.pdf> (acedido em 17/11/2025).
- Alliance, Bytecode (2025a). *Component Model Concepts - The WebAssembly Component Model*. url: <https://component-model.bytecodealliance.org/design/component-model-concepts.html> (acedido em 26/12/2025).
- (2026a). *ComponentizeJS: JavaScript toolchain for WebAssembly Components*. url: <https://github.com/bytecodealliance/ComponentizeJS> (acedido em 19/05/2026).
 - (2025b). *Components - The WebAssembly Component Model*. url: <https://component-model.bytecodealliance.org/design/components.html> (acedido em 26/12/2025).
 - (2025c). *Interfaces - The WebAssembly Component Model*. url: <https://component-model.bytecodealliance.org/design/interfaces.html> (acedido em 26/12/2025).
 - (2026b). *jco: JavaScript tooling for WebAssembly Components*. url: <https://bytecodealliance.github.io/jco/introduction.html> (acedido em 19/05/2026).
 - (2025d). *Why the Component Model? - The WebAssembly Component Model*. url: <https://component-model.bytecodealliance.org/design/why-component-model.html> (acedido em 26/12/2025).
 - (2025e). *Worlds - The WebAssembly Component Model*. url: <https://component-model.bytecodealliance.org/design/worlds.html> (acedido em 13/12/2025).
- Auditing package dependencies for security vulnerabilities | npm Docs* (2026). url: <https://docs.npmjs.com/auditing-package-dependencies-for-security-vulnerabilities> (acedido em 23/05/2026).
- Bellard, Fabrice e Charlie Gordon (2026a). *QuickJS JavaScript Engine*. url: <https://bellard.org/quickjs/> (acedido em 19/05/2026).

- Bellard, Fabrice e Charlie Gordon (2026b). *QuickJS JavaScript Engine Manual*. url: <https://bellard.org/quickjs/quickjs.html> (acedido em 19/05/2026).
- Besozzi, Valerio et al. (set. de 2025). *WebAssembly and Unikernels: A Comparative Study for Serverless at the Edge*. arXiv:2509.09400 [cs] version: 1. doi: 10.48550/arXiv.2509.09400. url: <http://arxiv.org/abs/2509.09400> (acedido em 11/11/2025).
- BlackDuck (2025). *2025 Open Source Security and Risk Analysis Report*. Rel. téc. url: <https://securityboulevard.com/2025/06/transitive-dependencies-are-exponentially-increasing-open-source-software-risk/>.
- Brown, Simon (2026). *Diagrams*. C4 model. url: <https://c4model.com/diagrams> (acedido em 29/06/2026).
- Cloudflare (fev. de 2025). *Security model*. en. url: <https://developers.cloudflare.com/workers/reference/security-model/> (acedido em 17/11/2025).
- Computing Machinery, Association for (2018). *ACM Code of Ethics and Professional Conduct*. url: <https://www.acm.org/code-of-ethics> (acedido em 14/12/2025).
- Cornelissen, Eric, Mikhail Shcherbakov e Musard Balliu (2024). «GHunter: Universal Prototype Pollution Gadgets in JavaScript Runtimes». en. Em: pp. 3693–3710. isbn: 978-1-939133-44-1. url: <https://www.usenix.org/conference/usenixsecurity24/presentation/cornelissen> (acedido em 30/12/2025).
- De Macedo, João et al. (jun. de 2022). «WebAssembly versus JavaScript: Energy and Runtime Performance». Em: *2022 International Conference on ICT for Sustainability (ICT4S)*, pp. 24–34. doi: 10.1109/ICT4S55073.2022.00014. url: <https://ieeexplore.ieee.org/document/9830108/> (acedido em 01/11/2025).
- Devographics (2025). *State of JavaScript 2025: Other Tools*. en. url: <https://2025.stateofjs.com/en-US/other-tools/> (acedido em 22/06/2026).
- ECMAScript® 2027 Language Specification (2026). url: <https://tc39.es/ecma262/> (acedido em 29/06/2026).
- Eriksen, Charlie (set. de 2025). *npm debug and chalk packages compromised*. url: <https://www.aikido.dev/blog/npm-debug-and-chalk-packages-compromised> (acedido em 12/06/2026).
- GitHub Advisory Database (2025). *Lodash Has a Prototype Pollution Vulnerability in `_`.`unset` and `_omit` Functions*. en. url: <https://github.com/advisories/GHSA-xxjr-mmjv-4gpg> (acedido em 15/06/2026).
- Google (2026a). *Chrome DevTools Protocol*. url: <https://chromedevtools.github.io/devtools-protocol/> (acedido em 29/06/2026).
- (2026b). *Chrome DevTools Protocol - Performance Domain*. url: <https://chromedevtools.github.io/devtools-protocol/tot/Performance/#method-getMetrics> (acedido em 29/06/2026).
- (2026c). *Design of V8 bindings*. url: https://chromium.googlesource.com/chromium/src/+master/third_party/blink/renderer/bindings/core/v8/V8BindingDesign.md#Isolate (acedido em 02/01/2026).
- Gruber, Jakob (14 de ago. de 2018). *Embedded Builtins*. en. url: <https://v8.dev/blog/embedded-builtins> (acedido em 08/06/2026).
- Haas, Andreas et al. (jun. de 2017). «Bringing the web up to speed with WebAssembly». en. Em: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Barcelona Spain: ACM, pp. 185–200. isbn: 978-1-4503-4988-8. doi: 10.1145/3062341.3062363. url: <https://dl.acm.org/doi/10.1145/3062341.3062363> (acedido em 05/11/2025).

- Harris, Charles R. et al. (set. de 2020). «Array programming with NumPy». Em: *Nature* 585.7825, pp. 357–362. issn: 1476-4687. doi: 10.1038/s41586-020-2649-2. url: <https://www.nature.com/articles/s41586-020-2649-2> (acedido em 29/06/2026).
- Houis, Tariq et al. (2026). «Bullseye: Detecting Prototype Pollution in NPM Packages with Proof of Concept Exploits». Em: *Proceedings 2026 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA, USA: Internet Society. isbn: 979-8-9919276-8-0. doi: 10.14722/ndss.2026.230211. url: <https://www.ndss-symposium.org/wp-content/uploads/2026-s211-paper.pdf> (acedido em 23/05/2026).
- Hunter, John D. (mai. de 2007). «Matplotlib: A 2D Graphics Environment». Em: *Computing in Science & Engineering* 9.3, pp. 90–95. issn: 1558-366X. doi: 10.1109/MCSE.2007.55. url: <https://ieeexplore.ieee.org/abstract/document/4160265> (acedido em 29/06/2026).
- Jiang, Zhuochen e Baojian Hua (jul. de 2025). «WasmSeps: Effectively Protecting WebAssembly Through Privilege Separation». Em: *2025 25th International Conference on Software Quality, Reliability and Security (QRS)*. ISSN: 2693-9177, pp. 703–714. doi: 10.1109/QRS65678.2025.00074. url: <https://ieeexplore.ieee.org/document/11173483/> (acedido em 02/11/2025).
- Kim, Minseo, Hyerean Jang e Youngjoo Shin (jul. de 2022). «Avengers, Assemble! Survey of WebAssembly Security Solutions». Em: *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. ISSN: 2159-6190, pp. 543–553. doi: 10.1109/CLOUD55607.2022.00077. url: <https://ieeexplore.ieee.org/document/9860829> (acedido em 01/11/2025).
- Kruchten, Philippe (2004). *The Rational Unified Process: An Introduction*. Google-Books-ID: RYCMx6o47pMC. Addison-Wesley Professional. 332 pp. isbn: 978-0-321-19770-2.
- Lazaroff, Jacob (2026). *quickjs-emscripten*. url: <https://github.com/justjake/quickjs-emscripten> (acedido em 19/05/2026).
- McKinney, Wes (2010). «Data Structures for Statistical Computing in Python». Em: *Proceedings of the 9th Python in Science Conference*. Ed. por Stéfan van der Walt e Jarrod Millman, pp. 56–61. doi: 10.25080/Majora-92bf1922-00a.
- MDN Web Docs (2026a). *Browser Compatibility tables and Browser Compatibility Data*. en. url: https://developer.mozilla.org/en-US/docs/MDN/Writing_guidelines/Page_structures/Compatibility_tables (acedido em 28/06/2026).
- (2026b). *Web APIs*. en. url: <https://developer.mozilla.org/en-US/docs/Web/API> (acedido em 28/06/2026).
- (2026c). *Window*. pt. url: <https://developer.mozilla.org/pt-BR/docs/Web/API/Window> (acedido em 28/06/2026).
- Microsoft (2026a). *Browsers | Playwright*. url: <https://playwright.dev/docs/browsers> (acedido em 29/06/2026).
- (2026b). *Using the Compiler API*. GitHub. url: <https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API> (acedido em 29/06/2026).
- Narayan, Shravan et al. (2020). «Retrofitting Fine Grain Isolation in the Firefox Renderer». en. Em: pp. 699–716. isbn: 978-1-939133-17-5. url: <https://www.usenix.org/conference/usenixsecurity20/presentation/narayan> (acedido em 05/11/2025).
- National Vulnerability Database (2026). *NVD - CVE-2026-22709*. url: <https://nvd.nist.gov/vuln/detail/CVE-2026-22709> (acedido em 29/06/2026).
- npm-audit | npm Docs* (2026). url: <https://docs.npmjs.com/cli/v11/commands/npm-audit> (acedido em 22/05/2026).

- package-lock.json* / *npm Docs* (2026). url: <https://docs.npmjs.com/cli/v11/configuring-npm/package-lock-json> (acedido em 22/05/2026).
- Porto, Instituto Politécnico do (nov. de 2020). *Regulamento do Código de Boas Práticas e de Conduta do Instituto Politécnico do Porto*. url: <https://diariodarepublica.pt/dr/detalhe/despacho/11171-2020-148327696> (acedido em 14/12/2025).
- Project, V8 (2026). *V8 JavaScript engine*. url: <https://v8.dev/> (acedido em 03/01/2026).
- Proxy - JavaScript* / *MDN* (17 de dez. de 2024). MDN Web Docs. url: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Proxy (acedido em 29/06/2026).
- Rayarao, Surya Rao e Naga Donikena (set. de 2025). «The Shai-Hulud NPM Supply Chain Attack: A Comprehensive Analysis of Self-Replicating Malware in the JavaScript Ecosystem». Em: url: <https://www.authorea.com/doi/full/10.22541/au.175830854.42750868?commit=9bc3206ca741614366ba5db6e82b14eba325b6d0> (acedido em 03/11/2025).
- Re:Earth (2026). *quickjs-emsripten-sync*. url: <https://github.com/reearth/quickjs-emsripten-sync> (acedido em 19/05/2026).
- Requiring 2FA for package publishing and settings modification* / *npm Docs* (2026). url: <https://docs.npmjs.com/requiring-2fa-for-package-publishing-and-settings-modification> (acedido em 22/05/2026).
- Romano, Alan et al. (mai. de 2022). «Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to WebAssembly». Em: *2022 IEEE Symposium on Security and Privacy (SP)*. ISSN: 2375-1207, pp. 1574–1589. doi: 10.1109/SP46214.2022.9833626. url: <https://ieeexplore.ieee.org/abstract/document/9833626> (acedido em 03/11/2025).
- Sharma, Varun (2025). *20+ Popular NPM Packages Compromised (Chalk, Debug, Strip-ANSI, Color-Convert, Wrap-ANSI...)* url: <https://www.stepsecurity.io/blog/20-popular-npm-packages-compromised-chalk-debug-strip-ansi-color-convert-wrap-ansi> (acedido em 12/06/2026).
- Shcherbakov, Mikhail, Paul Moosbrugger e Musard Balliu (mai. de 2024). «Unveiling the Invisible: Detection and Evaluation of Prototype Pollution Gadgets with Dynamic Taint Analysis». en. Em: *Proceedings of the ACM Web Conference 2024*. Singapore Singapore: ACM, pp. 1800–1811. isbn: 979-8-4007-0171-9. doi: 10.1145/3589334.3645579. url: <https://dl.acm.org/doi/10.1145/3589334.3645579> (acedido em 30/12/2025).
- Slavin, Ronen e Roni Kuznicki (set. de 2025). *npm debug / chalk Supply-Chain Attack: The Complete Guide*. url: <https://cycode.com/blog/npm-debug-chalk-supply-chain-attack-the-complete-guide/> (acedido em 12/06/2026).
- Sonatype (2024). *10th Annual State of the Software Supply Chain Report*. en-us. Rel. téc. Sonatype Inc., p. 69. url: [Sonatype%20Inc.](https://www.sonatype.com/state-of-the-software-supply-chain) (acedido em 25/11/2025).
- Staicu, Cristian-Alexandru, Michael Pradel e Benjamin Livshits (2018). «SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS». Em: *NDSS*. url: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_07A-2_Staicu_paper.pdf.
- TC39 (10 de fev. de 2025). *ShadowRealm Proposal Specification*. en. url: <https://tc39.es/proposal-shadowrealm/> (acedido em 08/06/2026).
- (2026a). *ShadowRealm Proposal*. en. url: <https://github.com/tc39/proposal-shadowrealm> (acedido em 08/06/2026).
 - (2026b). *ShadowRealm Proposal Explainer*. en. url: <https://github.com/tc39/proposal-shadowrealm/blob/main/explainer.md> (acedido em 08/06/2026).
- The Software House (2024). *State of Frontend 2024*. en. url: <https://tsh.io/state-of-frontend#libraries> (acedido em 22/06/2026).

- Trusted publishing for npm packages | npm Docs* (2026). url: <https://docs.npmjs.com/trusted-publishers> (acedido em 22/05/2026).
- V8 Project (2026a). *Getting Started with Embedding V8*. en. url: <https://v8.dev/docs/embed> (acedido em 08/06/2026).
- (2026b). *V8 API Reference: Isolate Class Reference*. en. url: https://v8.github.io/api/head/classv8_1_1Isolate.html (acedido em 08/06/2026).
- Verifying artifacts* (2026). SLSA. url: <https://slsa.dev/spec/v1.0/verifying-artifacts> (acedido em 23/05/2026).
- Verifying ECDSA registry signatures | npm Docs* (2026). url: <https://docs.npmjs.com/verifying-registry-signatures> (acedido em 23/05/2026).
- Viewing package provenance | npm Docs* (2026). url: <https://docs.npmjs.com/viewing-package-provenance> (acedido em 22/05/2026).
- Wasm 3.0 Completed - WebAssembly* (17 de set. de 2025). url: <https://webassembly.org/news/2025-09-17-wasm-3.0/> (acedido em 23/05/2026).
- WebAssembly* (2025). en. url: <https://webassembly.org/> (acedido em 02/11/2025).
- WebAssembly Community Group (2025). *WebAssembly Core Specification: Overview*. en. url: <https://webassembly.github.io/spec/core/intro/overview.html> (acedido em 08/06/2026).
- (2026). *Security*. en. url: <https://webassembly.org/docs/security/> (acedido em 08/06/2026).
- Webpack (2026a). *Compilation Object*. webpack. url: <https://webpack.js.org/api/compilation-object/#createchildcompiler> (acedido em 29/06/2026).
- (2026b). *Compiler Hooks*. webpack. url: <https://webpack.js.org/api/compiler-hooks/> (acedido em 29/06/2026).
- (2026c). *NormalModuleReplacementPlugin*. webpack. url: <https://docs.webpack.js.org/plugins/normal-module-replacement-plugin> (acedido em 29/06/2026).
- Wiegratz, Jasper Alexander (nov. de 2024). *Comparing Security and Efficiency of WebAssembly and Linux Containers in Kubernetes Cloud Computing*. arXiv:2411.03344 [cs]. doi: 10.48550/arXiv.2411.03344. url: <http://arxiv.org/abs/2411.03344> (acedido em 02/11/2025).
- Zahan, Nusrat et al. (2022). «What are weak links in the npm supply chain?» Em: *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '22. New York, NY, USA: Association for Computing Machinery, pp. 331–340. isbn: 978-1-4503-9226-6. doi: 10.1145/3510457.3513044. url: <https://doi.org/10.1145/3510457.3513044> (acedido em 02/11/2025).
- Zimmermann, Markus et al. (2019). «Small World with High Risks: A Study of Security Threats in the npm Ecosystem». en. Em: pp. 995–1010. isbn: 978-1-939133-06-9. url: <https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman> (acedido em 17/11/2025).

Apêndice A

Capacidades Web Candidatas a Exposição Controlada

Este anexo apresenta uma seleção representativa de capacidades Web que podem ser consideradas no modelo de exposição controlada da solução. A seleção tem como origem a documentação da MDN, em particular a referência geral de Web APIs, a documentação da interface `Window` e as tabelas de compatibilidade mantidas pela MDN (MDN Web Docs 2026a,b,c).

A tabela não substitui a lista completa de APIs Web mantida pela MDN. O seu objetivo é sintetizar capacidades particularmente relevantes para uma solução em que bibliotecas JavaScript/npm isoladas podem receber, por configuração, objetos ou funções do ambiente browser. A seleção privilegia APIs documentadas, capacidades acessíveis direta ou indiretamente através do ambiente global do browser e interfaces com relevância clara para segurança, privacidade, comunicação, armazenamento, identidade, acesso a dispositivos ou observabilidade.

Tabela A.1: Capacidades Web candidatas a exposição controlada

Capacidade	Categoria	Relevância
<code>fetch</code>	Rede e comunicação	Permite efetuar pedidos HTTP e pode ser usado para exfiltração de dados, comunicação com servidores externos ou interceção de tráfego aplicacional.
<code>XMLHttpRequest</code>	Rede e comunicação	Fornecer um canal alternativo de comunicação HTTP, ainda usado por código legado e relevante para ataques que interceptam ou modificam pedidos.
<code>WebSocket</code>	Rede e comunicação	Abre canais persistentes de comunicação bidirecional, com impacto direto na capacidade de enviar ou receber dados fora do fluxo HTTP tradicional.
<code>EventSource</code>	Rede e comunicação	Permite receber eventos enviados pelo servidor, podendo introduzir canais contínuos de entrada de dados para código de terceiros.
<code>navigator.sendBeacon</code>	Rede e comunicação	Permite enviar dados assíncronos de forma discreta, incluindo durante a saída da página, sendo sensível em cenários de telemetria ou exfiltração.
<code>localStorage</code>	Armazenamento	Disponibiliza armazenamento persistente por origem, podendo expor tokens, preferências, identificadores ou outros dados usados pela aplicação.
<code>sessionStorage</code>	Armazenamento	Disponibiliza armazenamento por sessão, relevante para dados temporários que continuam sensíveis dentro do ciclo de vida da página.
<code>indexedDB</code>	Armazenamento	Permite armazenar volumes maiores e estruturas mais complexas, podendo conter dados aplicacionais sensíveis ou estado offline.
<code>caches</code>	Armazenamento	Dá acesso ao armazenamento da Cache API, podendo afetar recursos reutilizados pela aplicação e cenários associados a service workers.
<code>document.cookie</code>	Armazenamento	Pode permitir leitura ou alteração de cookies acessíveis por JavaScript, com impacto direto em sessão, rastreamento e estado de autenticação quando não protegidos por atributos adequados.
<code>crypto</code>	Criptografia e identidade	Fornecer primitivas criptográficas e geração de aleatoriedade; pode ser necessária para bibliotecas legítimas, mas deve ser exposta com superfície reduzida quando possível.

Tabela A.1: Capacidades Web candidatas a exposição controlada (continuação)

Capacidade	Categoria	Relevância
credentials	Criptografia e identidade	A Credential Management API relaciona-se com credenciais e autenticação, devendo ser tratada como capacidade sensível.
PublicKeyCredential	Criptografia e identidade	Suporta autenticação forte baseada em WebAuthn, envolvendo identidade do utilizador e interação com autenticadores.
TextEncoder	Codificação e dados	Converte texto em bytes e pode ser necessária para bibliotecas que processam dados binários, sem conceder por si só acesso a recursos externos.
TextDecoder	Codificação e dados	Converte bytes em texto e é relevante para bibliotecas que processam imagens, ficheiros ou dados codificados.
Blob	Codificação e dados	Representa dados binários imutáveis e pode participar na construção de ficheiros, respostas, URLs locais ou payloads.
File	Codificação e dados	Representa ficheiros selecionados ou manipulados no browser, podendo carregar metadados e conteúdo fornecido pelo utilizador.
FileReader	Codificação e dados	Permite ler conteúdo de ficheiros no contexto da página, sendo relevante quando dependências processam dados fornecidos pelo utilizador.
FormData	Codificação e dados	Estrutura dados de formulários e ficheiros para transmissão, podendo conter dados pessoais ou credenciais aplicacionais.
URL	Codificação e dados	Permite construir e analisar URLs, incluindo URLs locais para objetos, influenciando fluxos de navegação, recursos e dados serializados.
URLSearchParams	Codificação e dados	Permite construir e interpretar parâmetros de URL, frequentemente usados para transportar estado, identificadores ou dados de consulta.
Intl	Internacionalização	Fornece formatação, calendários e dados de localização; é funcionalmente necessária em algumas bibliotecas, mas pode acrescentar chamadas frequentes ao ambiente hospedeiro.
document	DOM e navegação	Dá acesso à árvore DOM e ao estado da página, permitindo leitura, alteração visual, inserção de scripts ou manipulação de conteúdo aplicacional.

Tabela A.1: Capacidades Web candidatas a exposição controlada (continuação)

Capacidade	Categoria	Relevância
<code>location</code>	DOM e navegação	Expõe e pode alterar a localização da página, sendo relevante para redirecionamentos, leitura de parâmetros e navegação controlada por código.
<code>history</code>	DOM e navegação	Permite alterar o histórico de navegação e o estado visível da aplicação, afetando fluxos de interface e rotas em aplicações Web.
<code>customElements</code>	DOM e navegação	Permite registar componentes personalizados, podendo modificar a superfície de execução e interação do DOM.
<code>postMessage</code>	Comunicação entre contextos	Permite comunicação entre janelas, frames ou workers, podendo ser usado para transferir dados entre fronteiras de contexto.
<code>MessageChannel</code>	Comunicação entre contextos	Cria canais de mensagens dedicados, relevantes para comunicação estruturada entre contextos e transferência de capacidades.
<code>BroadcastChannel</code>	Comunicação entre contextos	Permite comunicação entre contextos da mesma origem, podendo propagar estado ou sinais entre separadores e janelas.
<code>Worker</code>	Comunicação entre contextos	Permite criar execução paralela em contexto separado, com implicações de desempenho, comunicação e controlo de código executado.
<code>SharedWorker</code>	Comunicação entre contextos	Permite partilhar um worker entre diferentes contextos da mesma origem, criando estado e comunicação persistente entre páginas.
<code>navigator.serviceWorker</code>	Comunicação entre contextos	Controla o registo e interação com service workers, que podem intercepar pedidos e gerir cache no âmbito da origem.
<code>navigator</code>	Dispositivos e permissões	Agrega várias capacidades do browser e informação do ambiente, incluindo possíveis vetores de fingerprinting e acesso indireto a APIs sensíveis.
<code>navigator.permissions</code>	Dispositivos e permissões	Permite consultar permissões do browser, expondo informação sobre capacidades autorizadas ou disponíveis para a origem.
<code>navigator.geolocation</code>	Dispositivos e permissões	Permite obter localização do utilizador mediante permissão, envolvendo dados pessoais sensíveis.

Tabela A.1: Capacidades Web candidatas a exposição controlada (continuação)

Capacidade	Categoria	Relevância
navigator.mediaDevices	Dispositivos e permissões	Dá acesso a dispositivos de áudio, vídeo e captura de ecrã mediante permissão, com elevado impacto em privacidade.
navigator.clipboard	Dispositivos e permissões	Permite ler ou escrever no clipboard em condições controladas pelo browser, podendo afetar dados copiados pelo utilizador.
Notification	Dispositivos e permissões	Permite apresentar notificações ao utilizador, dependendo de permissão e podendo ser usado para interação abusiva ou engenharia social.
PaymentRequest	Dispositivos e permissões	Interage com fluxos de pagamento do browser, envolvendo dados financeiros e decisões explícitas do utilizador.
performance	Observabilidade e desempenho	Expõe medições temporais e informação de desempenho, útil para avaliação mas também relevante para fingerprinting e canais laterais temporais.
PerformanceObserver	Observabilidade e desempenho	Permite observar eventos de desempenho, podendo revelar características da execução, carregamento de recursos e comportamento da página.