

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO PIAUÍ**

**CAMPUS TERESINA CENTRAL**

**CURSO ANÁLISE E DESENVOLVIMENTO DE SISTEMAS**

**GERMANO BARBOSA DA SILVA JÚNIOR**

***NOUDEJS*: *SCRIPTING* PARA AMBIENTES VIRTUAIS SERIALIZÁVEIS**

**TERESINA**

**2023**

GERMANO BARBOSA DA SILVA JÚNIOR

*NOUDEJS*: *SCRIPTING* PARA AMBIENTES VIRTUAIS SERIALIZÁVEIS

Trabalho de Conclusão de Curso (artigo científico) apresentado como exigência parcial para obtenção do diploma do Curso de Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia do Piauí, Campus Teresina Central .

Orientadora: Profª. Drª. Gabriela Meireles.

TERESINA

2023

***NOUDEJS*: *SCRIPTING* PARA AMBIENTES VIRTUAIS SERIALIZÁVEIS**

Germano Barbosa da Silva Júnior[[1]](#footnote-2)

Gabriela Meireles[[2]](#footnote-3)

**RESUMO**

Este artigo apresenta um interpretador especializado em JavaScript (ECMA 6), que tem como objetivo principal a interrupção da execução para a *serialização*, armazenamento e recuperação do estado de um sistema. Com essa base, busca-se criar ferramentas que possibilitem a simulação de hardware e suas características específicas dentro de ambientes virtualizados. Essa abordagem visa melhorar o aprendizado de conceitos fundamentais, como BIOS, boot, discos, volumes, partições, sistemas de arquivos, formatação de computadores, kernel, entre outros, de maneira intuitiva, sem a necessidade de acesso direto às tecnologias em questão.

As tecnologias mencionadas costumam ser negligenciadas devido à sua estabilidade e à existência de ferramentas que operam corretamente sem expor suas funcionalidades internas. No entanto, essa falta de conhecimento cria uma lacuna entre muitos técnicos, que, ao se depararem com problemas, optam por reinstalar todo o sistema a partir do zero, reconfigurando-o em cima da parte problemática, ao invés de solucionar o problema pontualmente, uma vez que desconhecem o seu funcionamento.

Com o intuito de resolver essa questão, este artigo propõe uma base para o desenvolvimento de uma ferramenta capaz de avançar nessa área de estudo, auxiliando na compreensão e na exploração dessas tecnologias. A partir do interpretador de JavaScript e da abordagem de interrupção da execução com estado serializável, espera-se fornecer aos profissionais e pesquisadores uma plataforma que permita a experimentação e o estudo aprofundado desses conceitos, contribuindo para o aprimoramento do conhecimento e da resolução de problemas nesse contexto.

**Palavras-chave:** interpretador de JavaScript, estado serializável, simulação de hardware, ambientes virtualizados, conhecimento técnico.

**ABSTRACT**

This article presents a specialized JavaScript interpreter (ECMA 6) whose main objective is to interrupt execution for serialization, storage, and recovery of a system's state. With this foundation, the goal is to create tools that enable the simulation of hardware and its specific characteristics within virtualized environments. This approach aims to enhance the learning of fundamental concepts such as BIOS, boot, disks, volumes, partitions, file systems, computer formatting, kernel, among others, in an intuitive manner, without the need for direct access to the technologies in question.

The mentioned technologies are often overlooked due to their stability and the existence of tools that operate correctly without exposing their internal functionalities. However, this lack of knowledge creates a gap among many technicians who, when faced with problems, choose to reinstall the entire system from scratch, reconfiguring it on top of the problematic part, instead of addressing the problem specifically, as they are unaware of how it functions.

With the aim of resolving this issue, this article proposes a foundation for the development of a tool capable of advancing in this area of study, assisting in the understanding and exploration of these technologies. By utilizing the JavaScript interpreter and the approach of interrupting execution with serializable state, it is hoped to provide professionals and researchers with a platform that enables experimentation and in-depth study of these concepts, contributing to the improvement of knowledge and problem-solving in this context.

**Keywords:** JavaScript interpreter, serializable state, hardware simulation, virtualized environments, technical knowledge, problem-solving.

Data de aprovação: 30/06/2023

# 1 Introdução

Este artigo apresenta um interpretador inovador de JavaScript, desenvolvido com o objetivo de oferecer suporte à interrupção da execução, *serialização*, armazenamento e recuperação do estado de um sistema. O foco principal está na criação e na análise desse interpretador, explorando sua capacidade de manipulação e gerenciamento do estado do programa. Embora o desenvolvimento de uma ferramenta abrangente para simulação de sistemas computacionais seja uma perspectiva futura, este artigo concentra-se exclusivamente no interpretador em si e nos detalhes relacionados à sua funcionalidade.

A linguagem de programação JavaScript foi escolhida como base para o interpretador devido à sua simplicidade, estabilidade, ampla utilização e flexibilidade na construção de aplicações web e outras soluções tecnológicas. No entanto, diferentemente dos interpretadores convencionais, nosso foco principal é a capacidade de interromper a execução do programa e, posteriormente, serializar e armazenar seu estado, permitindo sua recuperação posterior. Essa abordagem permite o estudo do sistema e sua interrupção para análise posterior, sem a necessidade da reconstrução do estado.

A capacidade de interromper a execução e serializar o estado do programa traz benefícios significativos para a análise detalhada e a depuração de sistemas complexos. Os estudantes e desenvolvedores podem pausar a execução em pontos específicos, examinar o estado atual do programa e retomar a simulação a partir desse ponto, facilitando a identificação e a correção de possíveis erros e comportamentos indesejados. Além disso, a *serialização* do estado permite a realização de experimentos controlados e repetíveis, contribuindo para a compreensão aprofundada do funcionamento interno dos sistemas, no contexto de simulação de hardware, isso permitiria que um estudante interrompesse o processo de *boot*, para uma análise mais aprofundada, com a capacidade de ver todos os componentes em operação, do terminal até o metal, diferentemente da máquina real, onde comumente é impossível interromper o *boot*, nas etapas primárias de inicialização.

Neste artigo, descreveremos a arquitetura do interpretador, incluindo os principais componentes e estruturas de dados utilizados. Apresentaremos as técnicas empregadas para interromper a execução do programa, serializar o estado e armazená-lo de forma eficiente. Também discutiremos os desafios enfrentados durante o desenvolvimento do interpretador e as soluções adotadas para garantir sua robustez e eficiência.

Embora a criação de uma ferramenta completa para simulação de sistemas computacionais seja uma perspectiva empolgante para o futuro, este artigo estabelece uma base sólida ao desenvolver um interpretador de JavaScript com recursos avançados de gerenciamento de estado. Acreditamos que essa abordagem pode contribuir significativamente para a análise e o estudo aprofundado de áreas da tecnologia das informações consideradas mais inacessíveis, para os quem tem menos iniciação na área.

**2 INTERPRETAÇÃO DO CÓDIGO**

O interpretador usa uma abordagem comum de compiladores, onde o código é analisado e convertido para um *byte-code*, uma representação binária das instruções que devem ser seguidas para obter o resultado da execução do programa. Para obter a *serialização*  desejada, basta que no momento de execução todos os componentes envolvidos na execução sejam *serializáveis.* Um componente é *serializável* quando seu estado pode ser representado como uma sequência de *bytes*, e então seu estado pode ser construído a partir de exclusivamente dos mesmos *bytes*, um exemplo de um sistema serializável é uma planilha, que contém dados que não dependem de nada fora da planilha, pode ser *serializado* para um arquivo, e então reconstruído de forma fiel, a partir do mesmo arquivo, um exemplo de um sistema que não é *serializável*, é uma partida de um jogo online, onde o sistema envolve a conexão com os computadores dos clientes, é impossível representar uma *conecção* *TCP/IP* em forma de bytes, tornando o sistema não *serializável*, de fato, jogos online comumente só podem ser salvos quando não existem jogadores online, por causa desse fato. Isso significa que o interpretador precisara que o seu ambiente de execução seja composto de componentes *serializáveis*, um desses componentes é o *byte-code*, que já é uma representação binária, no entanto o desafio é a *serialização* do estado da máquina.

**2.1 CONSTRUÇÃO DO *BYTE-CODE***

O módulo que constrói o *byte-code* segue os seguintes passos:

1. Leitura do código fonte
2. *Tokenização*: o processo de analisar os individuais caracteres do código e transformá-los em *tokens*.
3. Estruturação da sintaxe: processar os *tokens* produzidos nas estruturas de alto nível que formam o código, como afirmações, expressões e declarações.
4. Compilação: processar a estrutura gerada nas instruções que executam o que a estrutura deve fazer.

**2.1.1 LEITURA DO CÓDIGO FONTE**

Apesar de parecer trivial, uma oportunidade interessante nessa etapa é a leitura de todo o código de uma vez, e o carregamento de todo o código em uma região na memória que existirá durante toda a geração do buffer, isso permite que os próximos passos referenciem essa memória em vez de alocar novo espaço para guardar nomes e valores, já que estes já existem no código original, referenciar o código fonte em vez de obter uma cópia do valor, também traz a vantagem de permitir a inferência de qual código gerou qual estrutura, uma capacidade importante para a emissão de erros que exibem o código que gerou o erro.

**2.1.2 *TOKENIZAÇÃO***

Nessa etapa o código consiste apenas de *bytes*, que se encontram em alguma *encodificação* como *UTF-8*, ou *WIN-1252*, ou seja, podemos pensar no código fonte, não como uma sequência de bytes, mas como uma sequência de caracteres *Unicode*, a distinção é importante, pois o padrão ECMA 6, faz referência não á bytes, mas sim a caracteres *Unicode*, quando ele vai definir o que é espaço em branco, por exemplo.

Aqui está um exemplo de código a ser *tokenizado*:

function main() {

for (let n = 2; n < 100; n++) {

// faz uma operação filter

if (is\_prime(n)) {

console.log(n);

}

}

}

// retorna true, se n for primo

function is\_prime(n) {

for (let k = 2; k < n; k++) {

if (n % k === 0) {

// n é divisível por k e k é menor que n, com certeza n não é primo

return false;

}

}

return true;

}

main();

O processo de tokenização separa tokens, um token é uma sequência de letras, símbolos e números chamado de identificador ou palavra-chave, dependendo do texto, ou é um caractere especial como um parêntesis, ou uma chave, que tem significado especial na sintaxe, esses caracteres são chamados de pontuação, e sempre viram tokens individualmente, sem se juntar a outros tokens próximos.

Aqui está o código anteriormente exibido, já *tokenizado*, ele está representado como tokens separados por espaços, para o interpretador, isso é uma sequência de referências para o código fonte original.

function main ( ) { for ( let n = 2 ; n < 100 ; n + + ) { if ( is\_prime ( n ) ) { console . log ( n ) ; } } } function is\_prime ( n ) { for ( let k = 2 ; k < n ; k + + ) { if ( n % k = = = 0 ) { return false ; } } return true ; }

**2.1.2.1 ESPAÇO EM BRANCO**

Observe que toda a formação foi removida, pois para o computador ela não afeta como o código é executado, observe também como letras adjacentes continuam adjacentes, enquanto pontuação sempre fica em seu próprio token, também é interessante notar como os comentários foram removidos, pois são considerados espaço em branco.

Não é possível demonstrar aqui, mas na implementação do interpretador, cada token carrega metadados, sobre o espaço em branco que vem após o token, classificado em “junto”, quando não há espaço entre esse token e o próximo, “espaço” quando há espaço, e “linha” quando há uma quebra de linha após o token.

Esssa informação é importante para as próximas fases da compilação, pois, apesar de espaço em branco ser no geral desprezível, existem algumas condições em que ele não é, um desses casos, pode ser observado a seguir.

// Exemplo 1

return

a + b

// Exemplo 2

return a + b

No exemplo acima, a maioria interpretaria as duas linhas como, retornar a soma de a e b, no entanto, o padrão ECMA 6, diz que o primeiro exemplo, são duas linhas de código, primeiro uma instrução return, sem uma expressão, então uma expressão sozinha, o qual o valor será descartado.

Isso se dá pois existe um sistema chamado inserção automática de ponto e vírgula, que no interpretador apresentado, é implementado, usando o sistema de espaços mencionados. Após um return, nós esperamos ou uma expressão ou um ponto e vírgula, primeiro verificamos se há um ponto e vírgula, nesse caso não há, no entanto o espaço do último é do tipo “linha”, que tem o mesmo efeito e portanto terminamos a linha de código.

Esse foi o uso do metadado do espaço quando é do tipo “linha”, já o tipo “junto” é relevante para operadores.

// Exemplo 1

a ++ b

// Exemplo 2

a + + b

No exemplo 1 o dois mais são interpretados como um operador de incrementação, após ele, é esperado um operador infixo, ou seja um operador que vem entre dois valores ou o fim da expressão, no entanto encontramos b, o que emite um erro. Já no exemplo 2, o primeiro mais é interpretado como o operador infixo, portanto após ele, esperamos uma expressão, então encontramos o segundo operador mais, este é interpretado como um operador prefixo, que é um tipo de operador que espera um valor a sua direita, esse valor é procurado e obtemos o b, concluindo a análise da expressão.

Como se pode ver deste exemplo, o espaço em branco entre dois operadores, se presente ou ausente, pode alterar a forma como o código é compilado portanto é essencial que o interpretador leve isso em conta.

Esses exemplos são bem extremos, no entanto se o interpretador seguir o padrão nos detalhes, não haverão problemas quando chegar a hora de implementar novas funcionalidades.

**2.1.2.2 COMENTÁRIOS**

O comentário é uma porção do código com a finalidade de anotar informações pertinentes ao código onde se encontra, e não tem nenhum efeito na execução do código, a ausência ou presença destes não afetam de forma alguma o comportamento do programa, portanto o ideal é que estes sejam eliminados o quanto antes nas fases de compilação. A fase que foi determinada a ser mais apropriada para esta tarefa é a *tokenização*, pois está fase entende os individuais *tokens*, um algorítimo em uma fase anterior à *tokenização*, poderia por exemplo, eliminar comentários dentro de *strings*, o que seria um erro já que é impossível criar um comentário dentro de uma *string*.

A fase de *tokenização* funciona em amplos termos da seguinte forma, um ponteiro é inicializado para apontar para o primeiro caractere do *buffer* anteriormente criado, então caso o ponteiro esteja apontando para espaço em branco, o avançamos, e repetimos isso até que ele encontre algum código. Então quando quisermos obter o primeiro *token*, salvamos uma cópia deste ponteiro, e avançamos ele até que encontre o fim do *token*, o que geralmente significa espaço em branco, com isso teremos o começo e o fim do *token* atual, no entanto, nós repetimos a operação anterior, onde avançamos o ponteiro caso continue em espaço em branco, até que ele encontre o começo do próximo *token*, então dependendo de que tipo de espaço em branco foi encontrado, podemos determinado o metadado de espaço, se é “espaço”, “junto” ou “linha”, se uma quebra de linha tiver sido vista alguma hora, é tipo “linha”, se nenhum espaço em branco foi visto, então é “junto”, senão é “espaço”.

Neste algorítimo, podemos já remover os comentários de uma forma bem simples, nós ampliamos o conceito de espaço em branco para incluir os comentários de linha e de bloco, enquanto o ponteiro avança sobre o espaço em branco, caso ele encontre duas barras seguidas, ele se lembra que está em um comentário de linha e ignora todos os caracteres até encontrar uma quebra de linha, que termina esse comentário, e caso encontre uma barra seguida de um asterisco, ignora todos os caracteres até encontrar um asterisco seguido de uma barra.

**2.1.2.3 OTIMIZAÇÕES**

Uma característica desta fase, é que retorna referências ao buffer criado anteriormente, o que significa que até agora apenas uma grande alocação foi realmente feita, e uma otimização desta fase, é que não precisamos criar um vetor de *tokens*. Ao invés disso, criamos um objeto chamado um iterador, que sempre que pedimos ele retorna o próximo token, dessa forma, podemos usar os *tokens* um por um sem haver a necessidade de guardar todos em uma nova alocação. Caso precismos voltar para um certo ponto, podemos criar uma cópia deste iterador, e caso seja necessários continuamos a partir da cópia, o que resulta no mesmo efeito que se nós tivéssemos andado alguns *tokens* para trás, normalmente, voltar atrás não é possível em um iterador que só anda para frente, mas se simplesmente fizermos ele de tal forma que seja possível cloná-lo, então conseguimos essa habilidade sem quase nenhum custo.

**2.1.3 ESTRUTURAÇÃO DA SINTAXE**

Após a *tokenização*, os tokens são lidos em ordem, um por um, para determinar a estrutura do código, é nessa fase que erros de sintaxe são emitidos quando não é possível estruturar o código em termos de afirmações, expressões e declarações.

A análise de sintaxe, tenta estruturar o código em afirmações, uma afirmação, pode ser uma de três tipos:

1. Uma declaração, onde algo é criado como uma função ou variável
2. Uma expressão, onde um valor é calculado
3. Uma afirmação de controle de fluxo, como um *loop*, ou um *return*.

Essa análise, em amplos termos, funciona da seguinte forma: Se estamos procurando uma afirmação, e encontrarmos o token function, então o próximo token deve ser um identificador com o nome da função, se não houver um identificador, emite um erro, se houver, os próximos tokens devem definir a lista de argumentos, e assim por diante, após executar essa lógica até o fim, resulta na seguinte na estrutura:

função:  
 nome = main  
 parametros:  
 corpo:  
 for:  
 inicialização:  
 let:  
 nome = n  
 numero = 2  
 condição:  
 expressão:  
 menor que:  
 identificador = n  
 numero = 2  
 incremento:  
 expressão:  
 pos\_incremento:  
 identificador = n  
 corpo:  
 if:  
 condição:  
 chama:  
 função:  
 identificador = is\_prime  
 expressão:  
 identificador = n  
 corpo:  
 expressão:  
 chama:  
 função:  
 membro:  
 direita = log  
 esquerda:  
 identificador = console  
 expressão:  
 identificador = n

função:  
 nome = is\_prime  
 parâmetros:  
 identificador = n  
 corpo:  
 for:  
 inicialização:  
 let:  
 nome = k  
 número = 2  
 condição:  
 expressão:  
 menor que:  
 identificador = k  
 identificador = n  
 incremento:  
 expressão:  
 pos\_incremento:  
 identificador = k  
 corpo:  
 if:  
 condição:  
 identico:  
 mod:  
 identificador = n  
 identificador = k  
 número = 0  
 corpo:  
 return:  
 expressão:  
 false  
 return:  
 expressão:  
 true  
expressão:  
 chama:  
 função:  
 identificador = main

Essa estrutura pode então ser usada para a compilação.

**2.1.4 COMPILAÇÃO**

A diferença entre um interpretador e um compilador, é que o compilador faz um executável, enquanto o interpretar faz o executável, e imediatamente o executa. Portanto é natural que haja uma etapa de compilação em todo interpretador.

Nessa etapa, a estrutura anterior é analisada, e uma série de instruções simples são geradas que executam a lógica.

A seguir está a estrutura anterior mas compilada em instruções.

0 UNDEFINED 93 JMP -75

1 FUNCTION 8 98 POP

6 CALL 0 99 UNDEFINED

11 POP 100 RETURN

12 UNDEFINED 101 UNDEFINED

13 QUIT 102 INTEGER\_8 2

14 UNDEFINED 104 ASSIGN 0

15 INTEGER\_8 2 109 POP

17 ASSIGN 0 110 PUSH 0

22 POP 115 PUSH -1

23 PUSH 0 120 LESS

28 INTEGER\_8 100 121 JPF 35

30 LESS 126 PUSH -1

31 JPF 62 131 PUSH 0

36 PUSH 0 136 REMAINDER

41 UNDEFINED 137 ZERO

42 FUNCTION 54 138 IDENTICAL

47 CALL 1 139 JPF 6

52 JPF 30 144 FALSE

57 PUSH 0 145 RETURN\_POP 1

62 GLOBAL\_MEMBER "console" 150 POST\_INCREMENT 0

71 DUPE 155 POP

72 MEMBER "log" 156 JMP -51

81 CALL 1 161 POP

86 POP 162 TRUE

87 POST\_INCREMENT 0 163 RETURN

92 POP

O texto aqui presente é uma representação textual do *byte-code*, que é composto por bytes, antes de cada instrução existe um número que indica em qual posição do *byte-code* a instrução se encontra.

Com um pouco de paciência, é possível ver a relação entre o código fonte e o *byte-code*, de 0 a 13, existem as instruções que chamam a função main e terminam o programa, de 14 a 100 são as instruções da implementação da função *main*, e de 101 a 163 é a implementação da função *is\_prime*.

**2.2 EXECUÇÃO DO *BYTE-CODE***

O código é executado em uma máquina virtual, isso é, o *byte-code* não foi feito para ser executado em uma máquina real, como uma CPU, ao invés disso, o *byte-code* serve para uma CPU imaginária com limitações e especificações imaginárias, é a responsabilidade da máquina virtual prover esse ambiente em uma máquina real, para que o *byte-code* possa ser executado.

Essa abordagem de máquina virtual, é usada na linguagem Java, que tem máquinas virtuais (também conhecido como *JVM, Java Virtual Machine*) implementadas para vários dispositivos, por isso apenas o *byte-code*, distribuído como arquivos de extensão *jar*, é necessário para todos os dispositivos, desde que disponha de uma máquina virtual.

Normalmente em projetos baseados em outras linguagens como C/C++, o suporte a múltiplas arquiteturas e sistemas operacionais são feitos com compilação condicional, onde um mesmo código fonte tem que só são compiladas para certos ambientes, resultando em múltiplos binários, para cada ambiente.

A implementação escolhida, segue o princípio da *stack-machine* ou máquina de pilha, onde o estado da máquina se resumo a uma pilha, e as instruções operam nos valores do topo da pilha. A linguagem Python trabalha dessa maneira. A alternativa é uma *register-machine*, que é como a linguagem Lua funciona, onde as instruções operam com um número limitado de registradores, e acessam a pilha em pontos arbitrários, é dessa forma que as instruções de CPUs reais funcionam, esse modelo é no geral mais rápido que uma *stack-machine*, mas é mais complexo de implementar, motivo pelo qual Python não a usa, e esse projeto também não.

Por exemplo, em uma *stack-machine*, para somar um mais dois, primeiro o número um é adicionado a pilha, então o número dois é adicionado a pilha, e chamamos a instrução ADD, que pega os dois valores no topo da pilha, soma e adiciona no topo da pilha o valor somado.

Já em uma *register-machine*, para somar um mais dois, carregamos os valores para dois registradores livres, e chamamos a instrução *ADD*, e como argumentos para essa instrução, passamos os dois registradores usados e mais um registrador para guardar o resultado, toda *register-machine* dispõe de um número limitado de registradores, e a pilha continua presente, para o caso de todos os registradores estarem sendo usados e .

A linguagem JavaScript não usa um ou o outro, pois a linguagem JavaScript é apenas o padrão ECMA 6, e tem várias implementações no mercado, a implementação mais famosa é o V8 desenvolvido originalmente para o navegador *Chrome*, essa implementação usa *JIT*, ou compilação *Just In Time*, uma técnica de otimização fora do escopo deste projeto, onde as instruções *byte-code* são traduzidas para código de máquina real, resultado em uma velocidade de execução superior a qualquer máquina virtual de software.

**2.2.1 ESTADO DE MÁQUINA**

A intenção é construir uma máquina *serializável*, por isso, o módulo de execução foi construído com isso em mente, a ideia, é ter o estado do sistema como um objeto *serializável*, e então passamos o estado para uma função que retorna o novo estado da máquina, para executar a máquina até o fim, chamamos essa função de novo com o novo estado para obter o próximo estado, e repetimos isso até o programa terminar, que acontece quando a instrução QUIT é encontrada.

O estado da máquina é o coração do interpretador, ele é composto pelos seguintes componentes:

* Uma referência à próxima instrução a ser executada
* A pilha (uma pilha de valores e *frames*)
* O *frame* (valores contextuais a chamada da função atual)

No entanto, não há instruções que interajam com o mundo exterior, isso significa que a função que avança o estado da máquina é uma função pura, ou seja, ela é incapaz de causar um efeito colateral e incapaz e fazer algo útil, como escrever para o *console*. Para permitir que o código execute trabalho e interaja com o mundo exterior é preciso introduzir o conceito conhecido como *binding*.

**2.2.2 *BINDINGS***

Uma máquina virtual é inútil se não for capaz de comunicar-se com o mundo exterior. Por isso existe um conceito chamado *binding*.

Máquinas virtuais são feitas de tal forma que o programa sendo executado dentro dela não tenha acesso ao ambiente fora, por isso JavaScript, Java e ActionScript (popularizado pelo Adobe Flash Player) foram algumas das linguagens escolhidas no começo da internet para trazer as páginas da web a vida. No entanto, Java e ActionScript tinham muitas vulnerabilidades, onde o código dentro dela conseguia escapar da máquina virtual e chamar funções do sistema operacional para instalar *malwares* na máquina real e com o tempo o seu suporte foi sendo abandonado.

O *binding* mais bem conhecido do JavaScript é a função console.log, que escreve dados arbitrários formatados para o console.

É impossível implementar a console.log em JavaScript puro sem usar um outro *binding* para reproduzir a funcionalidade.

Normalmente, um *binding*, poderia ser implementado adicionando um ponteiro de função dentro do valor do objeto que pode ser chamado, esse ponteiro seria chamado, quando a máquina virtual tentasse chamar esse objeto como uma função, geralmente isso é como *bindings* são implementados. No entanto, se a máquina virtual for implementada dessa forma, isso a torna *inserializavel*, isso se dá por que ponteiros de função não podem ser serializados com segurança, pois ao armazenar o estado serializado e restaurar, a função pode se encontrar em outro lugar, por vários motivos, como distribuição aleatória de código no linux, ou a possibilidade do binário que executa a máquina virtual ter mudado, a deserialização de ponteiros de função também é uma falha de segurança propriamente dita, pois um estado em disco arquitetado maliciosamente pode ter um ponteiro para um local arbitrário, permitindo execução de código arbitrário.

A solução é usar um dicionário que mapeia números aos ponteiros de funções, e guardar na *serialização* os números no lugar dos ponteiros de função, na implementação, esse conceito, no interpretador foi chamado de *Native*, um identificador numérico que é mapeado aos ponteiros de funções. Isso resolve os problemas de segurança mencionados, pois o número não muda com diferentes executáveis, não importa se o sistema operacional mudar a localização das funções, pois o dicionário é nativo ao código e será atualizado como qualquer binário e é impossível criar um estado malicioso, pois um número inválido simplesmente não será encontrado no dicionário e isso resultará em um erro de *deserialização*, o que não é uma falha de segurança.

**3 CONCLUSÃO**

Em conclusão, este artigo estabeleceu uma base sólida para o desenvolvimento de um interpretador especializado, que pode ser uma valiosa ferramenta para aprimorar o entendimento de hardware no contexto educacional. O interpretador, conforme descrito neste artigo, demonstrou sua capacidade de executar scripts JavaScript, mantendo a seriabilidade do sistema em todos os momentos.

Ao introduzir essa abordagem, espera-se que o interpretador proposto desempenhe um papel fundamental na sala de aula, auxiliando alunos e professores no estudo e na exploração do funcionamento interno do hardware. Através dessa plataforma, os estudantes terão a oportunidade de obter uma compreensão prática e aprofundada dos conceitos fundamentais relacionados a hardware, como BIOS, *boot*, discos, volumes, partições, sistemas de arquivos, formatação de computadores e *kernels*.

Ao permitir que os alunos simulem e experimentem diretamente esses conceitos, a ferramenta proposta facilita a aprendizagem de forma intuitiva e interativa. Os estudantes poderão explorar diferentes cenários, testar configurações e observar os efeitos resultantes, desenvolvendo um conhecimento prático valioso que transcende a teoria pura.

Além disso, a manutenção da seriabilidade do sistema durante a execução dos scripts JavaScript é um recurso importante, pois permite que os alunos interrompam e salvem o estado do sistema em qualquer momento. Essa funcionalidade oferece a oportunidade de analisar e compreender o impacto das alterações nos diferentes componentes de hardware, proporcionando uma visão mais abrangente do funcionamento do sistema, impossível em hardware real.

Em suma, a base construída neste artigo para o desenvolvimento de um interpretador especializado em JavaScript, com a capacidade de manter a seriabilidade do sistema, apresenta um potencial significativo para avançar o entendimento de hardware na sala de aula. Essa ferramenta pode capacitar os estudantes a explorar, experimentar e compreender os conceitos fundamentais relacionados a hardware, bem como auxiliar os profissionais na resolução eficiente de problemas. Espera-se que essa abordagem inovadora contribua para o aprimoramento do ensino e do conhecimento prático em relação ao hardware, promovendo avanços significativos nesse campo vital da tecnologia.

REFERÊNCIAS

ECMA International. **ECMAScript® 2023 Language Specification**. Disponível em: <https://262.ecma-international.org/14.0/>. Acesso em: 28/06/2023.

Mozilla, JavaSript Reference MDN Disponível em: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/%5Bcaminho). Acesso em: 28/06/2023.

Germano Barbosa. **Noudejs**. Disponível em: https://github.com/kernel32dev/noude-js. Acesso em: 28/06/2023.

1. Estudante de análise e desenvolvimento de sistemas. IFPI. germanojunior007@gmail.com [↑](#footnote-ref-2)
2. Professora de metodologia científica, Gabriela Meireles. IFPI. [↑](#footnote-ref-3)