REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218

SEGURANÇA INTRÍNSECA EM IOT: COMPARATIVO DE RUST E C++ NA PREVENÇÃO DE VAZAMENTOS DE MEMÓRIA

INTRINSIC SAFETY IN IOT: A COMPARISON OF RUST AND C++ IN MEMORY LEAK PREVENTION

SEGURIDAD INTRÍNSECA EN IOT: COMPARACIÓN ENTRE RUST Y C++ EN LA PREVENCIÓN DE FUGAS DE MEMORIA

João Victor Souza[1], Wellington Marques da Silva[1], Jônatas Cerqueira Dias[2]

e676523
https://doi.org/10.47820/recima21.v6i7.6523

PUBLICADO: 7/2025

RESUMO

Este artigo investiga os mecanismos de segurança de memória oferecidos pela linguagem Rust em comparação com o C++ em aplicações de Internet das Coisas (IoT). A segurança de memória é um aspecto crítico em sistemas embarcados, onde falhas como *buffer overflow*, *double-free* e *use-after-free* podem comprometer a integridade e a disponibilidade das aplicações. Foram desenvolvidos e testados protótipos que simulam essas falhas em ambas as linguagens, permitindo observar como Rust lida com elas em tempo de compilação, enquanto o C++ permite sua execução mesmo sob análise com AddressSanitizer. Os mecanismos de *ownership*, *borrowing* e controle de *lifetimes* demonstraram-se eficazes na prevenção de falhas, compondo a base da segurança intrínseca do Rust. Apesar dos resultados promissores, reconhecem-se limitações como o uso de um ambiente controlado e ausência de testes com FFI e concorrência. Conclui-se que o Rust apresenta vantagens significativas em segurança de memória, sendo promissor para o desenvolvimento de sistemas IoT mais resilientes.

PALAVRAS-CHAVE: Rust. C++. Segurança intrínseca. IoT. Falhas de memória.

ABSTRACT

This article investigates the memory safety mechanisms provided by the Rust programming language in comparison to C++ in Internet of Things (IoT) applications. Memory safety is a critical aspect in embedded systems, where failures such as buffer overflow, double-free, and use-after-free may compromise the integrity and availability of applications. Prototype routines simulating these flaws were developed and tested in both languages to observe how Rust addresses them at compile time, whereas C++ allows their execution even under AddressSanitizer analysis. The ownership, borrowing, and lifetime control mechanisms proved effective in preventing failures, forming the basis of Rust's intrinsic safety model. Despite promising results, some limitations are acknowledged, such as the use of a controlled environment and the absence of tests involving FFI and concurrency. It is concluded that Rust presents significant advantages in memory safety, emerging as a promising alternative for developing more resilient IoT systems.

KEYWORDS: *Rust. C++. Intrinsic safety. IoT. Memory vulnerabilities.*

[1] Graduando em Análise e Desenvolvimento de Sistemas pela Faculdade de Tecnologia de Praia Grande.
[2] Doutorado em Engenharia de Controle e Automação Mecânica pela Escola Politécnica - Aplicação da Resiliência em Sistema de Controle Mecatrônico em Dispositivo de Assistência Ventricular. Mestrado em Engenharia de Software USP / IPT. Pós-graduação Lato Sensu em Análise de Sistemas. Pós-graduação Lato Sensu em Docência do Ensino Superior. Graduação em Licenciatura em Matemática. Graduação em Processamento de Dados. Faculdade de Tecnologia de Praia Grande.

ISSN: 2675-6218 - RECIMA21

REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218

RESUMEN
Este artículo investiga los mecanismos de seguridad de memoria ofrecidos por el lenguaje de programación Rust en comparación con C++ en aplicaciones de Internet de las Cosas (IoT). La seguridad de memoria es un aspecto crítico en los sistemas embebidos, donde fallos como desbordamiento de búfer, doble liberación (double-free) y uso después de liberar (use-after-free) pueden comprometer la integridad y disponibilidad de las aplicaciones. Se desarrollaron y probaron rutinas prototipo que simulan estos fallos en ambos lenguajes, permitiendo observar cómo Rust los bloquea en tiempo de compilación, mientras que C++ permite su ejecución incluso bajo análisis con AddressSanitizer. Los mecanismos de propiedad (ownership), préstamos (borrowing) y control de tiempos de vida (lifetimes) demostraron ser eficaces en la prevención de errores, formando la base de la seguridad intrínseca de Rust. A pesar de los resultados prometedores, se reconocen limitaciones experimentales. Se concluye que Rust ofrece ventajas significativas en seguridad de memoria, siendo una alternativa prometedora para sistemas IoT más resilientes.

PALABRAS CLAVE: *Rust. C++. Seguridad intrínseca. IoT. Fallos de memoria.*

INTRODUÇÃO

Nos últimos anos, a segurança no desenvolvimento de *software* tornou-se uma preocupação crítica para empresas e desenvolvedores em todo o mundo (Rezende, 2020). Com o aumento do número de dispositivos conectados e da complexidade dos sistemas, as vulnerabilidades, especialmente aquelas relacionadas ao gerenciamento de memória, têm gerado grandes prejuízos econômicos e riscos de segurança cibernética. Uma das falhas mais comuns e exploradas nesse contexto é o vazamento de memória, que pode ser explorado para comprometer a integridade de sistemas críticos (Riviera *et al.,* 2021).

Ataques que exploram erros de programação de memória, como estouros de *buffer*, continuam sendo uma das ameaças mais sérias à segurança (Akter *et al.,* 2023). O Centro de Coordenação do CERT (2021) lançou um documento que detalha uma vulnerabilidade no serviço de *spooler* de impressão do *Windows* que permite a execução remota de código arbitrário com privilégios de sistema. Essa vulnerabilidade, conhecida como *PrintNightmare*, exemplifica como falhas na gestão de memória podem ser exploradas por atacantes para comprometer sistemas. Embora não seja especificamente um estouro de *buffer*, trata-se de uma falha de segurança relacionada à manipulação inadequada de memória, reforçando a seriedade dessas ameaças. Os problemas de segurança de memória têm sido amplamente discutidos na literatura, especialmente em linguagens que oferecem baixo nível de controle, como C e C++. Moura (2023) aborda essas questões utilizando o contexto de programas *Kotlin*, evidenciando sua relevância para o desempenho dos sistemas.

Entre as linguagens de programação utilizadas para o desenvolvimento de *software*, o C++ destaca-se por sua flexibilidade e controle direto sobre o hardware, características especialmente valorizadas em sistemas embarcados e aplicações da Internet das Coisas (IoT),

REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218

devido à sua eficiência no uso de recursos. No entanto, essa liberdade vem acompanhada de riscos, pois o gerenciamento manual de memória em C++ frequentemente resulta em falhas como *buffer overflows*, *use-after-free* e *double-free*, responsáveis por até 70% das vulnerabilidades de segurança descobertas em grandes sistemas, como o Windows e o Firefox (Riviera *et al.,* 2021). Apesar de alguns considerarem a discussão sobre segurança de memória em linguagens de baixo nível como C++ e Rust superada, ela permanece extremamente relevante no atual cenário tecnológico. O crescimento da IoT (Godoi; Araújo, 2019) ampliou a necessidade de linguagens eficientes e próximas ao *hardware*, e o C++ continua a dominar esse ecossistema. No entanto, os desafios de segurança persistem, tornando importante a análise de alternativas como o Rust, que busca eliminar falhas comuns de gerenciamento de memória (Riviera *et al.,* 2021). Assim, compreender as diferenças entre essas linguagens não apenas permite o desenvolvimento de aplicações mais seguras, mas também ajuda a definir os rumos da evolução tecnológica em sistemas embarcados e IoT.

Introduzido pela Mozilla em 2010, Rust foi projetado para resolver os problemas mais comuns relacionados à segurança de memória, por meio de seu rigoroso sistema de *ownership* e *borrowing*, que elimina automaticamente vulnerabilidades como *use-after-free* e *buffer overflows* (Riviera *et al.,* 2021). O Rust se torna uma alternativa moderna ao C++ com o propósito de oferecer uma segurança intrínseca, sem comprometer o desempenho (Bugden, Alahmar, 2022). Nesse contexto, surge a seguinte questão de pesquisa: “Quais mecanismos de segurança implementados no Rust são mais eficazes na prevenção de vazamentos de memória em aplicações de IoT, em comparação aos métodos tradicionais usados no C++?”

Este estudo tem como objetivo verificar e comparar os mecanismos de segurança oferecidos pelo Rust em relação às práticas tradicionais de gerenciamento de memória do C++, com ênfase na prevenção de vazamentos de memória em dispositivos e sistemas IoT, avaliando o impacto dessas abordagens na segurança das aplicações.

RUST E C++ NA SEGURANÇA DE MEMÓRIA

A segurança de memória é um dos aspectos críticos no desenvolvimento de *software,* especialmente em sistemas que exigem baixo nível de abstração, alto desempenho e controle direto do *hardware*, como ocorre em aplicações IoT e sistemas embarcados. A crescente conectividade desses dispositivos amplia a superfície de ataque, tornando vulnerabilidades de memória um risco significativo para a segurança cibernética e operacional (Mullen; Meany, 2019).

O C++ tem sido uma das linguagens predominantes nesses domínios devido à sua flexibilidade e eficiência, permitindo um controle refinado sobre a alocação de recursos. No entanto, essa liberdade também introduz desafios de segurança, pois o gerenciamento manual da memória torna a linguagem suscetível a falhas como buffer *overflow*, *use-after-free* e *double-free*.

ISSN: 2675-6218 - RECIMA21

REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218

Tais vulnerabilidades já foram amplamente exploradas em incidentes relevantes, como no caso do *worm Code Red*, que afetou servidores da Microsoft ao possibilitar a execução remota de código por meio da sobrescrita de memória (Alenezi *et al.,* 2020; Damasio, 2022; Sousa, 2022). Além disso, erros de *use-after-free*, como os que comprometeram o Internet Explorer, evidenciam os riscos graves associados à manipulação inadequada de memória em aplicações C++ (Sousa, 2022).

Em contrapartida, o Rust surge como uma alternativa moderna e segura para substituir o C++ em ambientes de alto risco, incluindo dispositivos embarcados e sistemas de IoT. Seu sistema de *ownership* e *borrowing* elimina erros comuns de gerenciamento de memória já em tempo de compilação, reduzindo drasticamente a exposição a falhas críticas. Além disso, o Rust oferece segurança sem a necessidade de um coletor de lixo, garantindo previsibilidade e eficiência, características essenciais para aplicações em tempo real e para dispositivos com recursos computacionais limitados. Rebert e Kern (2024) reforçam que a segurança de memória é um requisito fundamental para o desenvolvimento de *software* seguro, destacando o potencial do Rust para mitigar vulnerabilidades, especialmente em sistemas críticos.

A seguir, são detalhadas as abordagens adotadas por Rust e C++ no gerenciamento de memória, ressaltando suas implicações para a segurança de aplicações em sistemas embarcados e na Internet das Coisas.

RUST E C++ NA SEGURANÇA: DIFERENTES ABORDAGENS DE GERENCIAMENTO DE MEMÓRIA

As linguagens Rust e C++ são bastante utilizadas no desenvolvimento de *software* de alto desempenho, mas adotam estratégias diferentes para o gerenciamento de memória. De acordo com a Microsoft (2023), o C++ oferece flexibilidade notável no acesso direto ao *hardware*, permitindo um controle eficiente dos recursos. Contudo, essa liberdade exige que o programador gerencie manualmente a alocação e liberação da memória, o que pode resultar em vulnerabilidades como *buffer overflow, use-after-free* e *double-free* (Damasio, 2022; Microsoft, 2023).

Embora melhorias como o uso de *smart pointers* tenham sido implementadas para fortalecer a segurança, o C++ ainda depende fortemente da disciplina do programador, o que mantém a possibilidade de erros críticos. Em contraste, o Rust impõe regras de posse e transferência de memória em tempo de compilação, prevenindo problemas como *use-after-free* e *double-free* de maneira automática, sem necessidade de intervenção manual.

Ataques envolvendo *buffer overflow* continuam sendo um dos principais riscos em aplicações C++, como evidenciado no caso do *worm Code Red* (Damasio, 2022; Souza, 2022; Alenezi *et al.,* 2020). Vulnerabilidades de *use-after-free*, como as que comprometeram o Internet

ISSN: 2675-6218 - RECIMA21

Explorer, também demonstram os perigos associados à gestão manual da memória (Sousa, 2022).

Ao contrário do C++, o Rust foi projetado para solucionar essas falhas estruturais desde sua concepção. Seu sistema de gerenciamento de memória, baseado nos conceitos de *ownership* e *borrowing*, assegura a liberação automática da memória quando ela não é mais necessária, prevenindo vazamentos e erros como *double-free*. Além disso, o *borrow checker*, atuando em tempo de compilação, impede o uso de ponteiros pendentes e condições de *use-after-free* (Ivanov, 2022; Bugden; Alahmar, 2022).

Outro diferencial importante é que, enquanto o C++ permite estouros de *buffer* se o programador não implementar verificações manuais, o Rust incorpora verificações automáticas para garantir que os limites de *buffers* não sejam ultrapassados. Ademais, o Rust elimina o risco de ponteiros nulos ao utilizar o tipo *Option*, prática que reduz consideravelmente uma das principais causas de falhas em C++ (Klabnik; Nichols, 2023).

Essas diferenças tornam o Rust uma linguagem mais robusta e segura para aplicações críticas em sistemas embarcados e na Internet das Coisas, onde a integridade e a confiabilidade dos dispositivos são importantes.

SEGURANÇA DE MEMÓRIA EM RUST: *OWNERSHIP, BORROWING* E VERIFICAÇÕES AUTOMATIZADAS

O sistema de *ownership* no Rust estabelece que cada valor possui um único proprietário, e a memória associada a esse valor é liberada automaticamente quando o proprietário sai de escopo (Klabnik; Nichols, 2023). Isso elimina a necessidade de mecanismos de coleta de lixo comuns em outras linguagens, mantendo a eficiência de execução e oferecendo segurança de memória sem comprometer o desempenho (Bugden; Alahmar, 2022). Além disso, o sistema de *borrowing* permite que referências sejam passadas de forma segura, prevenindo condições de corrida (*data races*) por meio de restrições aplicadas em tempo de compilação (Silva, 2023).

As verificações de segurança no Rust ocorrem majoritariamente em tempo de compilação, o que representa um diferencial em relação ao C++ e outras linguagens que dependem de verificações em tempo de execução. O *borrow checker* impede o uso de ponteiros mutáveis compartilhados, e o uso de *lifetimes* garante que referências não possam existir além do tempo de vida dos dados referenciados, prevenindo erros de *use-after-free* (Bugden; Alahmar, 2022; Silva, 2023). Além disso, o sistema de tipos do Rust impede o uso de ponteiros nulos, minimizando uma das causas mais comuns de falhas em C++ (Klabnik; Nichols, 2023).

Essas funcionalidades são essenciais para a segurança de memória no Rust e foram projetadas para garantir que muitos dos erros mais comuns em linguagens como C e C++ sejam impossíveis de ocorrer (Silva, 2023).

ISSN: 2675-6218 - RECIMA21

REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218

A INTERNET DAS COISAS E OS DESAFIOS DE SEGURANÇA DE MEMÓRIA

A Internet das Coisas (IoT) representa uma das maiores transformações tecnológicas contemporâneas, promovendo a conexão entre dispositivos físicos e sistemas computacionais para a coleta, o processamento e a troca de dados em tempo real. Essa integração abrange desde sensores industriais e equipamentos hospitalares até eletrodomésticos, veículos e dispositivos vestíveis. No entanto, à medida que esses sistemas se tornam mais onipresentes, crescem também os desafios relacionados à segurança, especialmente no que se refere à proteção da memória e à prevenção de falhas críticas (El-Sofany *et al.,* 2024).

Dispositivos IoT operam frequentemente com recursos limitados de processamento, energia e memória, o que restringe a implementação de técnicas tradicionais de segurança. Segundo Oliveira e Moreira (2021), os sistemas operacionais desses dispositivos precisam ser bastante otimizados para garantir funcionalidade mínima com baixo consumo, o que implica a exclusão de camadas convencionais de abstração e segurança. Essa restrição amplia a exposição a vulnerabilidades como *buffer overflow*, *stack smashing* e *use-after-free* — falhas comuns em sistemas programados em linguagens como C e C++, que não oferecem proteção intrínseca à memória (Oliveira; Moreira, 2021).

A análise de sistemas operacionais como Contiki, TinyOS, FreeRTOS e RIOT-OS revela que essas plataformas, apesar de serem muito utilizadas no universo IoT, apresentam vulnerabilidades recorrentes. Uma das falhas mais identificadas é o *stack smashing*, um tipo específico de *buffer overflow* que ocorre quando a pilha — área da memória usada para armazenar informações sobre chamadas de funções — é sobrecarregada. No contexto do TinyOS, essa falha pode permitir a execução de código malicioso ao sobrescrever regiões de memória, comprometendo a integridade do dispositivo (Oliveira; Moreira, 2021). Essas vulnerabilidades se tornam ainda mais perigosas em redes heterogêneas e amplamente distribuídas, onde não há garantias consistentes de privacidade e segurança das comunicações entre dispositivos (Oliveira; Moreira, 2021).

Outro aspecto importante refere-se à carência de padrões abertos para a distribuição de chaves criptográficas entre dispositivos, o que dificulta a criação de arquiteturas unificadas de segurança. Como apontado por Oliveira e Moreira (2021), essa ausência de padronização contribui para a fragmentação das soluções e aumenta o risco de exposição a ataques. Além disso, falhas como *buffer overflows* em servidores DNS e a inexistência de verificação de limites em datagramas ICMP já foram exploradas com sucesso em dispositivos baseados em RIOT-OS, demonstrando a gravidade dos riscos associados à ausência de proteção robusta de memória (Oliveira; Moreira, 2021).

ISSN: 2675-6218 - RECIMA21

REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218

Dhar *et al.,* (2024) reforçam esse cenário ao afirmar que os dispositivos IoT são alvos ideais para ataques justamente por causa de sua arquitetura leve e da escassez de mecanismos automatizados de defesa em profundidade. A combinação entre alta conectividade e baixa capacidade de defesa cria um ambiente propício à exploração de vulnerabilidades críticas. Nesse contexto, a adoção de linguagens como Rust, que impõem, em tempo de compilação, garantias formais de segurança de memória, representa uma alternativa para o desenvolvimento de aplicações em IoT.

Diante desse cenário, pensar em segurança desde o começo do desenvolvimento, o *by design,* é essencial quando se trata de dispositivos IoT. A proteção da memória, em especial, não pode ser deixada para depois. Ela precisa ser considerada logo nas primeiras camadas da arquitetura do sistema, levando em conta as limitações e particularidades desses dispositivos embarcados, que muitas vezes têm pouca memória, baixo processamento e pouca margem para erro.

MÉTODO

Esta pesquisa é de natureza bibliográfica e experimental, com o objetivo de analisar comparativamente como Rust e C++ tratam falhas de segurança de memória, especialmente em aplicações voltadas para dispositivos IoT. A revisão teórica foi realizada em bases como Google Scholar e SciELO, buscando estudos que abordam *buffer overflows*, *use-after-free*, *double-free* e os mecanismos de segurança implementados pelas duas linguagens. Os termos utilizados nas buscas estão apresentados na Tabela 1.

Além da revisão, foram realizados testes práticos por meio da construção de programas simples em Rust e C++ que simulam vulnerabilidades comuns de memória. Esses códigos foram executados em ambiente controlado, permitindo observar o comportamento das linguagens diante das falhas simuladas. A análise centrou-se nos mecanismos de prevenção automáticos do Rust, como *ownership* e *borrowing*, em contraste com a gestão manual exigida no C++.

Todos os testes foram realizados em um *desktop* ASUS com processador AMD Zen 2 (8 núcleos, 4,0 GHz) e 16 GB de RAM, executando Windows 10 Home 22H2 (build 19045). Sobre esse *host* utilizou-se Windows Subsystem for Linux 2 (WSL 2) configurado com Ubuntu 24.04.1 LTS (kernel 6.6.87.2-microsoft-standard-WSL2). A compilação do C++ pelo GCC 13.3.0 (pacote Ubuntu 13.3.0-6ubuntu2~24.04), otimização -O2 e instrumentação de segurança via AddressSanitizer ativada pela flag *-fsanitize=address*, que detecta em tempo de execução falhas como *buffer overflow*, *use-after-free* e *double-free*. Já do Rust foi pelo rustc 1.86.0, gerenciada pelo cargo 1.86.0, em perfil --release, sem trechos *unsafe*.

Para assegurar reprodutibilidade com recursos limitados, foram avaliadas apenas três classes de falhas (*buffer overflow*, *use-after-free* e *double-free*) 22 em programas *monothread*,

ISSN: 2675-6218 - RECIMA21

sem bibliotecas externas nem chamadas FFI (*Foreign Function Interface*). O ambiente restringe-se a um único *hardware* x86-64 sob WSL 2/Ubuntu 24.04; microcontroladores ARM ou RISC-V, RTOS e contêineres não foram contemplados. Utilizaram-se versões atuais de compilador (GCC 13.3.0 + ASan e rustc 1.86.0) com otimização moderada; *builds* legados, *flags* agressivas (-O3, LTO) e binários *debug* ficaram fora do escopo. As métricas coletadas limitam-se a tempo total de execução, pico de RAM e tamanho do binário, sem medir consumo energético ou latência de I/O.

Tabela 1.Lista de Descritores de Busca de Artigos nas Plataformas Acadêmicas

Item	Termos-chave	Operador lógico
1	"*Buffer overflow*", "*use-after-free*", "*double-free*"	AND
2	"*Ownership*", "*Borrowing*"	AND
3	"Rust", "C++"	AND
4	"Internet das Coisas", "IoT", "sistemas embarcados"	AND
5	*"Security",* "segurança de memória"	AND
6	"Ponteiros inteligentes"	AND

Fonte: Elaborado pelos autores

RESULTADOS

Foram desenvolvidos e testados códigos em C++ e Rust que simulam três vulnerabilidades clássicas de memória: *buffer overflow, use-after-free* e *double-free*. As três rotinas-protótipo foram implementadas em C++ 17 e em Rust 1.76, reproduzindo os mesmos fluxos lógicos de uma aplicação de telemetria de sensores IoT (leitura, processamento e descarte de pacotes). Cada versão C++ foi compilada com *AddressSanitizer* (ASan) -O2; a contraparte Rust foi compilada em *release profile* padrão (sem código *unsafe*). Os códigos-fonte e os *prints* da tela de execução, foram numerados na tabela 2, e as métricas quantitativas resultantes estão consolidadas na Tabela 3.

Tabela 2. Resultados: Vulnerabilidades de Memória Simuladas em C++ e Rust

Vulnerabilidade simulada	Lingua-gem	Detecção	Momento da detecção	Saída/Erro	Execução prosse-gue?
Buffer overflow	C++	ASan	Runtime	SIGABRT + relatório ASan (quadro 1)	Não
	Rust	Compilador + panic	Runtime (*index check*)	*thread panicked* (quadro 4)	Não

ISSN: 2675-6218 - RECIMA21

REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218

Double free	C++	ASan	Runtime	SIGABRT + relatório ASan (quadro 3)	Não
	Rust	Compilador	*Compile-time* (*move semantics*)	E0382 "*value used after move*" (quadro 6)	—
Use-after-free	C++	ASan	Runtime	SIGSEGV (quadro 2)	Não
	Rust	Compilador	Compile-time *(borrow checker)*	E0505 "*borrow later used here*" (quadro 5)	—

Fonte: Elaborado pelos autores

Tabela 3 – Métricas quantitativas dos experimentos em C++ e Rust

Caso	Lingua gem	Estado	Tempo total (ms)	Pico de RAM (MB)	Tamanho binário (B)
Buffer-overflow	C++	OK (ASan detecta em runtime)	31 030	67	26 072
	Rust	OK (*panic* em runtime)	1 090	97	4 004 328
Use-after-free	C++	OK (ASan)	4 360	68	24 960
	Rust	Falha de compilação	N/A	N/A	N/A
Double-free	C++	OK (ASan)	800	64	26 192
	Rust	Falha de compilação	N/A	N/A	N/A

Fonte: Elaborado pelos autores

Código 1. Trecho em C++ evidenciando *buffer overflow* em leitura de sensores

```
#include <iostream>
#include <string.h>
using namespace std;

void processarLeitura(char* leitura) {
    char buffer[16];
    strcpy(buffer, leitura);
    cout << "Leitura processada: " << buffer << endl;
}

int main() {
    char leituraRecebida[] = "Temperatura: 28.7°C - Pressão: 1013hPa";
    processarLeitura(leituraRecebida);
    return 0;
}
```

Fonte: Elaborado pelos autores

ISSN: 2675-6218 - RECIMA21

REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218

Quadro 1. Execução do Código *buffer overflow* em C++

```
joao@DESKTOP-G4RUAAM:~/codigos/cpp$ g++ sensor_buffer_overflow.cpp -o sensor_overflow -
fsanitize=address
joao@DESKTOP-G4RUAAM:~/codigos/cpp$ ./sensor_overflow
=================================================================
==9762==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fa229900030 at pc
0x7fa22bfd0923 bp 0x7ffe8f5b4d90 sp 0x7ffe8f5b4538
WRITE of size 41 at 0x7fa229900030 thread T0
    #0 0x7fa22bfd0922 in strcpy ../../../../src/libsanitizer/asan/asan_interceptors.cpp:563
    #1 0x5601fc2fe32c in processarLeitura(char*) (/home/joao/codigos/cpp/sensor_overflow+0x132c) (BuildId:
e01b1cd2e747af5fd2e11b64eb2c96a497b888aa)
    #2 0x5601fc2fe528 in main (/home/joao/codigos/cpp/sensor_overflow+0x1528) (BuildId:
e01b1cd2e747af5fd2e11b64eb2c96a497b888aa)
    #3 0x7fa22b9ac1c9 in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
    #4 0x7fa22b9ac28a in __libc_start_main_impl ../csu/libc-start.c:360
    #5 0x5601fc2fe1c4 in _start (/home/joao/codigos/cpp/sensor_overflow+0x11c4) (BuildId:
e01b1cd2e747af5fd2e11b64eb2c96a497b888aa)

Address 0x7fa229900030 is located in stack of thread T0 at offset 48 in frame
    #0 0x5601fc2fe298 in processarLeitura(char*) (/home/joao/codigos/cpp/sensor_overflow+0x1298)
(BuildId: e01b1cd2e747af5fd2e11b64eb2c96a497b888aa)

  This frame has 1 object(s):
    [32, 48) 'buffer' (line 6) <== Memory access at offset 48 overflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind mechanism, swapcontext
or vfork
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow ../../../../src/libsanitizer/asan/asan_interceptors.cpp:563 in
strcpy
Shadow bytes around the buggy address:
  0x7fa2298ffd80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7fa2298ffe00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7fa2298ffe80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7fa2298fff00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7fa2298fff80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x7fa229900000: f1 f1 f1 f1 00 00[f3]f3 00 00 00 00 00 00 00 00
  0x7fa229900080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7fa229900100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7fa229900180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7fa229900200: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7fa229900280: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==9762==ABORTING
joao@DESKTOP-G4RUAAM:~/codigos/cpp$
```

Fonte: Elaborado pelos autores

ISSN: 2675-6218 - RECIMA21

REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218

Código 2. Trecho em C++ evidenciando *use-after-free* na atualização de dados de sensor

```
#include <iostream>
#include <string>
using namespace std;

void atualizarLeitura(string* dadoSensor) {
    delete dadoSensor; // libera o ponteiro
    cout << "Novo dado: " << *dadoSensor << endl; // use-
after-free
}

int main() {
    string* leitura = new string("28.5°C");
    atualizarLeitura(leitura);
    return 0;
}
```

Fonte: Elaborado pelos autores

Quadro 2. Execução do Código *use-after-free* em C++

```
joao@DESKTOP-G4RUAAM:~/codigos/cpp$ g++ sensor_use_after_free.cpp -o
sensor_use_after_free -fsanitize=address
joao@DESKTOP-G4RUAAM:~/codigos/cpp$ ./sensor_use_after_free
AddressSanitizer:DEADLYSIGNAL
=================================================================
==11282==ERROR: AddressSanitizer: SEGV on unknown address 0x00000000000c (pc
0x7fa658fa6b04 bp 0x7fffe5fecad0 sp 0x7fffe5feca90 T0)
==11282==The signal is caused by a READ memory access.
==11282==Hint: address points to the zero page.
    #0 0x7fa658fa6b04 in _IO_new_file_xsputn libio/fileops.c:1219
    #1 0x7fa658fa6b04 in _IO_new_file_xsputn libio/fileops.c:1197
    #2 0x7fa658f99a11 in __GI__IO_fwrite libio/iofwrite.c:39
    #3 0x7fa659538f18 in fwrite
../../../../src/libsanitizer/sanitizer_common/sanitizer_common_interceptors.inc:1109
    #4 0x7fa659392dc3 in std::basic_ostream<char, std::char_traits<char> >&
std::__ostream_insert<char, std::char_traits<char> >(std::basic_ostream<char,
std::char_traits<char> >&, char const*, long) (/lib/x86_64-linux-gnu/libstdc++.so.6+0x156dc3)
(BuildId: ca77dae775ec87540acd7218fa990c40d1c94ab1)
    #5 0x5562d89c9542 in atualizarLeitura(std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> >*) (/home/joao/codigos/cpp/sensor_use_after_free+0x2542) (BuildId:
c3b6f2d04768c8d2c84341b3cb38e7dd4776563c)
    #6 0x5562d89c965d in main (/home/joao/codigos/cpp/sensor_use_after_free+0x265d) (BuildId:
c3b6f2d04768c8d2c84341b3cb38e7dd4776563c)
    #7 0x7fa658f3d1c9 in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
    #8 0x7fa658f3d28a in __libc_start_main_impl ../csu/libc-start.c:360
    #9 0x5562d89c9424 in _start (/home/joao/codigos/cpp/sensor_use_after_free+0x2424) (BuildId:
c3b6f2d04768c8d2c84341b3cb38e7dd4776563c)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV libio/fileops.c:1219 in _IO_new_file_xsputn
==11282==ABORTING
joao@DESKTOP-G4RUAAM:~/codigos/cpp$
```

Fonte: Elaborado pelos autores

ISSN: 2675-6218 - RECIMA21

REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218

Código 3. Trecho em C++ evidenciado *double-free* em descarte de pacotes de sensor

```
#include <iostream>
using namespace std;

void descartarPacote(int* pacote) {
    delete pacote; // 1ª liberação
}

int main() {
    int* leituraPacote = new int(1024); // simula ID do pacote
    descartarPacote(leituraPacote);
    delete leituraPacote; // double-free
    return 0;
}
```

Fonte: Elaborado pelos autores

Quadro 3. Execução do Código *double-free* em descarte de pacotes de sensor em C++

```
joao@DESKTOP-G4RUAAM:~/codigos/cpp$ g++ sensor_double_free.cpp -o
sensor_double_free -fsanitize=address
joao@DESKTOP-G4RUAAM:~/codigos/cpp$ ./sensor_double_free
=================================================================
==10498==ERROR: AddressSanitizer: attempting double-free on 0x502000000010 in thread T0:
    #0 0x7fefc2eb55e8 in operator delete(void*, unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:164
    #1 0x560cc548f2aa in main (/home/joao/codigos/cpp/sensor_double_free+0x12aa) (BuildId:
fa72cb14d8cfd4cc6ef9013aa980350e737e36ba)
    #2 0x7fefc28391c9 in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
    #3 0x7fefc283928a in __libc_start_main_impl ../csu/libc-start.c:360
    #4 0x560cc548f144 in _start (/home/joao/codigos/cpp/sensor_double_free+0x1144) (BuildId:
fa72cb14d8cfd4cc6ef9013aa980350e737e36ba)

0x502000000010 is located 0 bytes inside of 4-byte region [0x502000000010,0x502000000014)
freed by thread T0 here:
    #0 0x7fefc2eb55e8 in operator delete(void*, unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:164
    #1 0x560cc548f22e in descartarPacote(int*)
(/home/joao/codigos/cpp/sensor_double_free+0x122e) (BuildId:
fa72cb14d8cfd4cc6ef9013aa980350e737e36ba)
    #2 0x560cc548f294 in main (/home/joao/codigos/cpp/sensor_double_free+0x1294) (BuildId:
fa72cb14d8cfd4cc6ef9013aa980350e737e36ba)
    #3 0x7fefc28391c9 in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
    #4 0x7fefc283928a in __libc_start_main_impl ../csu/libc-start.c:360
    #5 0x560cc548f144 in _start (/home/joao/codigos/cpp/sensor_double_free+0x1144) (BuildId:
fa72cb14d8cfd4cc6ef9013aa980350e737e36ba)

previously allocated by thread T0 here:
    #0 0x7fefc2eb4548 in operator new(unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:95
    #1 0x560cc548f247 in main (/home/joao/codigos/cpp/sensor_double_free+0x1247) (BuildId:
fa72cb14d8cfd4cc6ef9013aa980350e737e36ba)
    #2 0x7fefc28391c9 in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
    #3 0x7fefc283928a in __libc_start_main_impl ../csu/libc-start.c:360
```

ISSN: 2675-6218 - RECIMA21

REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218


```
    #4 0x560cc548f144 in _start (/home/joao/codigos/cpp/sensor_double_free+0x1144) (BuildId:
fa72cb14d8cfd4cc6ef9013aa980350e737e36ba)

SUMMARY: AddressSanitizer: double-free ../../../../src/libsanitizer/asan/asan_new_delete.cpp:164
in operator delete(void*, unsigned long)
==10498==ABORTING
joao@DESKTOP-G4RUAAM:~/codigos/cpp$
```

Fonte: Elaborado pelos autores

Código 4. Trecho em Rust evidenciando *buffer overflow* em leitura de sensores

```
fn processar_leitura(leitura: &str) {
  let mut buffer = [0u8; 16]; // buffer fixo
  let bytes = leitura.as_bytes();

  for i in 0..bytes.len() {
      buffer[i] = bytes[i]; // Erro que impede a execução
  }

  let resultado = String::from_utf8_lossy(&buffer);
  println!("Leitura processada: {}", resultado);
}

fn main() {
  let leitura_recebida = "Temperatura: 28.7°C - Pressão: 1013hPa";
  processar_leitura(leitura_recebida);
}
```

Fonte: Elaborado pelos autores

Quadro 4. Execução do Código *buffer overflow* em Rust

```
joao@DESKTOP-G4RUAAM:~/codigos/rust$ rustc sensor_buffer_overflow.rs -o
sensor_overflow_rust
joao@DESKTOP-G4RUAAM:~/codigos/rust$ ./sensor_overflow_rust

thread 'main' panicked at sensor_buffer_overflow.rs:6:7:
index out of bounds: the len is 16 but the index is 16
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
joao@DESKTOP-G4RUAAM:~/codigos/rust$
```

Fonte: Elaborado pelos autores

Código 5. Trecho em Rust evidenciando *use-after-free* em atualização de dados de sensor

```
fn main() {
  let leitura = String::from("28.5°C");
  let ponteiro = &leitura;

  drop(leitura); // Rust deixa compilar, mas qualquer uso de ponteiro
depois disso será bloqueado

  println!("Dado: {}", ponteiro); // Erro de compilação: borrowed value
after move
}
```

Fonte: Elaborado pelos autores

ISSN: 2675-6218 - RECIMA21

REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218

Quadro 5. Execução do Código *use-after-free* em Rust

```
joao@DESKTOP-G4RUAAM:~/codigos/rust$ rustc sensor_use_after_free.rs -o
sensor_use_after_free
error[E0505]: cannot move out of `leitura` because it is borrowed
 --> sensor_use_after_free.rs:5:8
  |
2 |   let leitura = String::from("28.5°C");
  |       ------- binding `leitura` declared here
3 |   let ponteiro = &leitura;
  |                  -------- borrow of `leitura` occurs here
4 |
5 |   drop(leitura); // ← Rust ainda deixa compilar, mas qualquer uso de `ponteiro` depois disso
será barrado
  |        ^^^^^^^ move out of `leitura` occurs here
6 |
7 |   println!("Dado: {}", ponteiro); // ← Erro de compilação: borrowed value after move
  |                        -------- borrow later used here
  |
help: consider cloning the value if the performance cost is acceptable
  |
3 |   let ponteiro = &leitura.clone();
  |                         ++++++++

error: aborting due to 1 previous error

For more information about this error, try `rustc --explain E0505`.
joao@DESKTOP-G4RUAAM:~/codigos/rust$
```

Fonte: Elaborado pelos autores

Código 6. Trecho em Rust evidenciando *double-free* em descarte de pacotes de sensor

```
fn descartar_pacote(pacote: Box<i32>) {
  println!("Pacote descartado: {}", pacote);
  // Aqui o pacote será automaticamente liberado quando sair do
escopo
}

fn main() {
  let pacote = Box::new(1024);
  descartar_pacote(pacote);
  descartar_pacote(pacote); //Erro: o pacote já foi movido
}
```

Fonte: Elaborado pelos autores

ISSN: 2675-6218 - RECIMA21

REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218

Quadro 6. Execução do Código *double-free* em Rust

```
joao@DESKTOP-G4RUAAM:~/codigos/rust$ rustc sensor_double_free.rs -o sensor_double_free
error[E0382]: use of moved value: `pacote`
 --> sensor_double_free.rs:9:20
  |
7 |   let pacote = Box::new(1024);
  |       ------ move occurs because `pacote` has type `Box<i32>`, which does not implement the
`Copy` trait
8 |   descartar_pacote(pacote);
  |                    ------ value moved here
9 |   descartar_pacote(pacote); // ← Erro: o pacote já foi movido
  |                    ^^^^^^ value used here after move
  |
note: consider changing this parameter type in function `descartar_pacote` to borrow instead if
owning the value isn't necessary
 --> sensor_double_free.rs:1:29
  |
1 | fn descartar_pacote(pacote: Box<i32>) {
  |    ----------------         ^^^^^^^^ this parameter takes ownership of the value
  |    |
  |    in this function
help: consider cloning the value if the performance cost is acceptable
  |
8 |   descartar_pacote(pacote.clone());
  |                          ++++++++

error: aborting due to 1 previous error

For more information about this error, try `rustc --explain E0382`.
joao@DESKTOP-G4RUAAM:~/codigos/rust$
```

Fonte: Elaborado pelos autores

DISCUSSÃO

Esses resultados mostram objetivamente como Rust e C++ diferem na abordagem de vulnerabilidades críticas no contexto IoT, com Rust oferecendo verificações automáticas em tempo de compilação ou execuções controladas, enquanto o C++ identifica os erros apenas durante a execução do programa.

Os resultados apontam uma vantagem intrínseca do modelo de segurança do Rust em prevenir vulnerabilidades críticas antes mesmo da execução do software, por meio do *borrow checker* e *ownership*. Isso contrasta com o C++, onde as vulnerabilidades só foram detectadas em tempo de execução, mesmo com o uso de ferramentas avançadas como *Address Sanitizer* (ASan).

De acordo com Safronov *et al.,* (2024), vulnerabilidades baseadas em memória constituem a maioria das ameaças em dispositivos IoT, indicando a importância de abordagens seguras por projeto, como as adotadas pelo Rust, capazes de prevenir tais falhas.

ISSN: 2675-6218 - RECIMA21

REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218

Os achados vão ao encontro com o estudo de Noseda *et al.,* (2022), que apontam a prevalência de vulnerabilidades de memória em sistemas baseados em C/C++ e a significativa redução dessas vulnerabilidades ao migrar para Rust. Estudos da *Cybersecurity and Infrastructure Security Agency* (CISA) e outras agências internacionais reforçam ainda mais essa perspectiva ao sugerir a adoção de linguagens seguras como forma definitiva para eliminar classes inteiras de vulnerabilidades (Cybersecurity and Infrastructure Security Agency, 2023).

Embora os testes tenham oferecido insights importantes, as limitações incluem a simplicidade dos cenários testados, ausência de concorrência e uso limitado de recursos de hardware específicos a dispositivos IoT reais, que podem apresentar vulnerabilidades adicionais ou comportamentos diferentes em cenários práticos.

A implicação prática é que a adoção de Rust pode potencialmente reduzir os riscos associados a ataques cibernéticos e vulnerabilidades em larga escala, especialmente em sistemas IoT onde atualizações frequentes são difíceis e custosas.

A linguagem Rust é reconhecida por sua segurança intrínseca de memória, alcançada via seu modelo de *ownership* e *borrowing*, prevenindo vulnerabilidades em tempo de compilação (Noseda *et al.*, 2022; Bugden; Alahmar, 2022; Silva, 2023). No entanto, sua adoção em dispositivos IoT de baixo nível é desafiadora, pois muitos operam com sistemas legados em C/C++, como Contiki, RIOT-OS ou FreeRTOS (Oliveira; Moreira, 2021). A interoperabilidade com esses sistemas exige o uso de FFI, mas, ao interagir com código C, as garantias de segurança do Rust não se estendem, reintroduzindo riscos de memória (Noseda *et al.*, 2022). Isso frequentemente demanda o uso de blocos *unsafe*, transferindo a responsabilidade da segurança ao desenvolvedor (Bugden; Alahmar, 2022).

Além disso, a maturidade do ecossistema do Rust, embora em crescimento, ainda é mais jovem que a de C++ (Bugden; Alahmar, 2022; Silva, 2023). Isso pode resultar na falta de bibliotecas prontas para casos de uso específicos ou para interação com *hardware* muito particular em dispositivos IoT com recursos limitados (Noseda *et al.*, 2022; Silva, 2023). A ausência de ferramentas de *build* e suporte a *targets* tradicionais de compilação no mercado de sistemas embarcados pode exigir que se escreva mais código de baixo nível, o que frequentemente envolve o uso de *unsafe* para comunicação direta com o *hardware* (Silva, 2023).

Além das falhas bloqueadas, os dados quantitativos da Tabela 3 permitem uma análise mais ampla da viabilidade prática do Rust em comparação ao C++. Em termos de tempo total de execução, foi observado que os binários C++ apresentaram tempos ligeiramente inferiores, o que é esperado devido à ausência de verificações internas. Já o Rust introduziu um pequeno *overhead*, principalmente nas simulações com *panics*, embora os tempos permaneçam aceitáveis para ambientes de teste.

ISSN: 2675-6218 - RECIMA21

REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218

O pico de uso de RAM foi similar entre as linguagens na maioria dos casos, com leve vantagem para o Rust apenas na simulação de *buffer overflow*. Quanto ao tamanho dos binários, os resultados foram divergentes: enquanto o Rust gerou arquivos menores na simulação de *use-after-free*, foi superado pelo C++ nos outros dois testes. Esses dados mostram que, embora o Rust apresente ganhos em segurança, seu custo em desempenho e *footprint* varia conforme o cenário, reforçando a necessidade de avaliação criteriosa conforme os requisitos de cada aplicação IoT.

CONSIDERAÇÕES

Este estudo buscou responder à pergunta: “Quais mecanismos de segurança implementados no Rust são mais eficazes na prevenção de vazamentos de memória em aplicações de IoT, em comparação aos métodos tradicionais usados no C++?”. A análise experimental de rotinas vulneráveis demonstrou que o Rust impede a construção de binários com falhas como *buffer overflow*, *use-after-free* e *double-free*, bloqueando essas vulnerabilidades em tempo de compilação ou gerando *panics* controlados. Em contraste, os mesmos códigos escritos em C++ puderam ser executados mesmo com falhas ativas, mesmo quando compilados com o AddressSanitizer. A dependência do C++ em práticas manuais e verificações em tempo de execução manteve-se suscetível a *buffer overflows*, *double free* e *use after free*, confirmando a literatura que atribui a essas falhas cerca de 70 % das *Common Vulnerabilities and Exposure* (CVEs) em sistemas embarcados (Noseda *et al.*, 2022).

Os mecanismos mais eficazes identificados no Rust incluem o sistema de *ownership*, que regula a posse e a liberação da memória; o modelo de *borrowing*, que evita acessos concorrentes inseguros; e o controle de *lifetimes*, que impede o uso de referências inválidas e a eliminação de ponteiros nulos e aritmética insegura, exceto sob uso declarado de *unsafe*, os quais delegam ao desenvolvedor a responsabilidade pelo controle manual. Esses elementos compõem o núcleo da chamada segurança intrínseca, um conjunto de garantias embutidas na própria linguagem e no compilador, não dependente de verificações externas.

Contudo, é importante reconhecer que os testes foram realizados em ambiente controlado, com execução *single-threaded*, em um único sistema operacional, sem uso de bibliotecas externas via FFI, ponto que frequentemente exige o uso de *unsafe*. Não foram abordadas falhas concorrentes (*data-races*), nem mensurados o consumo de energia ou o *footprint* completo de memória, aspectos relevantes no contexto de IoT real. Portanto, propõem se como trabalhos futuros a realização de novos estudos envolvendo sistemas mais complexos e testes em hardware real de dispositivos IoT, além de análises de desempenho e consumo de recursos em comparação direta com linguagens tradicionais.

ISSN: 2675-6218 - RECIMA21

Ao evidenciar ganhos de segurança e apontar lacunas a investigar, este trabalho contribui como passo inicial para uma agenda de pesquisa que viabilize uma IoT mais resiliente, orientando desenvolvedores, arquitetos e decisores sobre o caminho para sistemas embarcados mais seguros.

REFERÊNCIAS

AKTER, Mst S. *et al.* Feature engineering-based detection of buffer overflow vulnerability in source code using neural networks. *In:* **2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)**. IEEE, 2023. p. 765-776. Disponível em: https://ieeexplore.ieee.org/abstract/document/10196986. Acesso em: 21 set. 2024.

ALENEZI, M. N. *et al.* Evolution of Malware Threats and Techniques: a Review**. International Journal of Communication Networks and Information Security (IJCNIS)**, *[S. l.],* v. 12, n. 3, 2022. doi: 10.17762/ijcnis.v12i3.4723. Disponível em: https://www.ijcnis.org/index.php/ijcnis/article/view/4723. Acesso em: 20 out. 2024.

BUGDEN, W.; ALAHMAR, A. Rust: the programming language for safety and performance. *In:* **2nd International Graduate Studies Congress (IGSCONG'22)**, Turkey, 2022. Disponível em: https://doi.org/10.48550/arXiv.2206.05503. Acesso em: 20 out. 2024.

CERT - COORDINATION CENTER**. Microsoft Windows Print Spooler allows for RCE via AddPrinterDriverEx()**. Vulnerability Note VU#383432. Carnegie Mellon University, 2021. Disponível em: https://www.kb.cert.org/vuls/id/383432. Acesso em: 20 out. 2025.

CYBERSECURITY AND INFRASTRUCTURE SECURITY AGENCY. **The case for memory-safe roadmaps**: why both C-suite executives and technical experts need to take memory-safe coding seriously. Washington, D.C.: CISA, 2023. Disponível em: https://www.cisa.gov/resources-tools/resources/case-memory-safe-roadmaps. Acesso em: 12 abr. 2025.

DAMASIO, V. M. A. **Buffer overflow: mecanismo e exploração**. 2022. Trabalho de Conclusão de Curso (Graduação em Engenharia de Computação) – Pontifícia Universidade Católica de Goiás, Goiânia, 2022. Disponível em: https://repositorio.pucgoias.edu.br/jspui/handle/123456789/4433. Acesso em: 28 set. 2024.

DHAR, S. *et al.* Securing IoT devices: A novel approach using blockchain and quantum cryptography. **Internet of Things**, v. 25, 101960, 2024. Disponível em: https://www.sciencedirect.com/science/article/pii/S2542660523003426. Acesso em: 19 mar. 2025.

EL-SOFANY, H. *et al.* Using machine learning algorithms to enhance IoT system security. **Scientific Reports**, v. 14, n. 1, 2024. Disponível em: https://www.nature.com/articles/s41598-024-62861-y. Acesso em: 09 abr. 2025.

GODOI, M. G. de; ARAÚJO, L. S. A Internet das Coisas: evolução, impactos e benefícios. **Revista Interface Tecnológica**, Taquaritinga, SP, v. 16, n. 1, p. 19–30, 2019. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/538. Acesso em: 2 mar. 2025.

HUNTLEY, R. **Is C/C++ memory safe?** EE Times Europe. 2024. Disponível em: https://www.eetimes.eu/is-c-c-memory-safe/. Acesso em: 26 abr. 2025.

ISSN: 2675-6218 - RECIMA21

REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218

IVANOV, Nikolay. Is rust c++-fast? benchmarking system languages on everyday routines. **arXiv preprint arXiv:2209.09127**, 2022. Disponível em: https://arxiv.org/abs/2209.09127. Acesso em: 21 set. 2024.

KLABNIK, S.; NICHOLS, C. **The Rust Programming Language**. 2. ed. San Francisco: No Starch Press, 2023.

MICROSOFT. **C++ Documentation**. [S. l.]: Microsoft, 2023. Disponível em: https://learn.microsoft.com/pt-br/cpp/cpp/?view=msvc-170. Acesso em: 28 set. 2024.

MICROSOFT. **Ponteiros inteligentes (C++ moderno)**. [S. l.]: Microsoft, 2023. Disponível em: https://learn.microsoft.com/pt-br/cpp/cpp/smart-pointers-modern-cpp?view=msvc-170. Acesso em: 2 mar. 2025.

MOURA, D. Y. R. **Investigando as relações entre problemas de segurança de memória e desempenho em programas Kotlin**. 2023. Trabalho de Conclusão de Curso (Graduação em Engenharia Elétrica – Telecomunicações) – Faculdade de Tecnologia, Universidade Federal do Amazonas, Manaus, 2023. Disponível em: http://riu.ufam.edu.br/handle/prefix/6647. Acesso em: 21 set. 2024

MULLEN, G.; MEANY, L. Assessment of buffer overflow based attacks on an IoT operating system. *In:* **2019 Global IoT Summit (GIoTS)**. IEEE, 2019. p. 1-6. doi: 10.1109/GIOTS.2019.8766434. Disponível em: https://ieeexplore.ieee.org/document/8766434. Acesso em: 11 jan. 2025

NOSEDA, M. *et al.* Rust for secure iot applications: why c is getting rusty. *In:* **Embedded World Conference, Nuremberg, Germany, 21-23 June 2022**. WEKA, 2022. Disponível em: https://digitalcollection.zhaw.ch/handle/11475/25046. Acesso em: 06 maio 2025.

OLIVEIRA, L. R. M. de; MOREIRA, A. L. S. **Padrões de segurança para dispositivos IoT de baixo nível:** uma revisão comparativa. 2021. Trabalho de Conclusão de Curso (Graduação em Análise e Desenvolvimento de Sistemas) – Instituto Federal de Pernambuco, Recife, 2021. Disponível em: https://repositorio.ifpe.edu.br/xmlui/handle/123456789/664. Acesso em: 29 abr. 2025.

REBERT, A.; KERN, C. Secure by design: Google's perspective on memory safety. **Technical report. Google Security Engineering**, 2024. Disponível em: https://research.google/pubs/secure-by-design-googles-perspective-on-memory-safety/. Acesso em: 19 jan. 2025.

REZENDE, M. V. **Avaliação de segurança cibernética no desenvolvimento de software embarcado automotivo**: uma abordagem ontológica. 2020. Tese (Doutorado em Sistemas de Informação e Gestão do Conhecimento) - Universidade FUMEC, Belo Horizonte, 2020. Disponível em: https://repositorio.fumec.br/handle/123456789/835. Acesso em: 22 set. 2024.

RIVIERA, E *et al.* Keeping safe rust safe with galeed. *In:* **Proceedings of the 37th Annual Computer Security Applications Conference**. 2021. p. 824-836. Disponível em: https://doi.org/10.1145/3485832.3485903. Acesso em: 21 set. 2024.

SAFRONOV, V. *et al.* How memory-safe is iot? assessing the impact of memory-protection solutions for securing wireless gateways. In: **Proceedings of the 14th International Conference on the Internet of Things**. 2024. p. 261-266. Disponível em: https://ora.ox.ac.uk/objects/uuid:5627d10d-b7b2-46a7-895e-013a540c0973. Acesso em: 06 maio 2025.

ISSN: 2675-6218 - RECIMA21

REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218

SILVA, T. D. da. **CORAL**: a Rust-like Borrow Checker for C. 2023. Dissertação (Mestrado em Engenharia Informática e de Computação) – Faculdade de Engenharia, Universidade do Porto, Porto, 2023. Disponível em: https://hdl.handle.net/10216/153606. Acesso em: 06 out. 2024.

SOUSA, D. F. de. **AST-Based Large-Scale Vulnerability Analysis for C/C++**. [*S. l.: s. n.*], 2022. Disponível em: https://hdl.handle.net/10216/144794. Acesso em: 20 out. 2024.

ISSN: 2675-6218 - RECIMA21