УДК 004.77.052:004.056

# ЗАБЕЗПЕЧЕННЯ НАДІЙНОСТІ ТА БЕЗПЕКИ IOT-ПРИСТРОЇВ: ПЕРЕХІД ВІД C/C++ ДО RUST НА РІВНІ МІКРОПРОГРАМ

Ситник Є. С.
e-mail: yehor.sytnyk@nure.ua
Науковий керівник – асистент кафедри ПІ, Дашенков Д. С.
Харківський національний університет радіоелектроніки, каф. ПІ
м. Харків, Україна

This work analyzes the impact of memory management errors on IoT fault tolerance, justifying the transition from C/C++ to Rust for firmware. While C/C++ introduces risks like memory corruption and undefined behavior – causing vulnerabilities and crashes – Rust eliminates these at compile time via its ownership and borrowing models. Furthermore, Rust prevents exploits by turning buffer overruns into controlled panics, statically guarantees thread safety, and reduces cyclomatic complexity. Integrating Rust neutralizes prevalent memory vulnerabilities during compilation, radically improving hardware fault tolerance.

Сучасні IoT-пристрої часто інтегровані у критичну інфраструктуру, що вимагає їхньої безперебійної автономної роботи. Традиційним стандартом розробки прошивок залишаються мови C та C++. Попри їхню високу продуктивність, ручне управління пам'яттю створює архітектурні ризики: витоки, пошкодження даних та непередбачену поведінку. Це є першопричиною як вразливостей перед кібератаками, так і фатальних збоїв під час обробки нестандартної телеметрії, що робить програмну відмовостійкість критичним фактором життєздатності системи.

Метою роботи є аналіз впливу помилок управління пам'яттю на відмовостійкість IoT-пристроїв та обґрунтування переходу на мову Rust для створення стабільних електронних апаратів та програмних систем.

Яскравим прикладом наслідків таких архітектурних рішень є нещодавній аналіз вразливостей корпоративних VoIP-телефонів Grandstream (зокрема, CVE-2026-2329)[1]. Технічною причиною критичного збою стало використання небезпечної функції strcpy для обробки вхідних мережевих JSON-запитів. Дані копіювалися у малий 64-байтний буфер на стеку без будь-якої перевірки їхньої довжини.

В IoT-апаратурі через жорсткі обмеження ресурсів чи використання застарілих інструментальних ланцюгів базові механізми боротьби (такі як Stack Canaries, ASLR, PIE тощо) здебільшого відсутні. Як наслідок, відсутність перевірки меж буфера перетворює пристрій на легку мішень – цілеспрямований запит дозволяє зловмиснику отримати віддалене виконання коду (RCE) з правами “root”, тоді як випадковий нестандартний пакет даних просто призводить до пошкодження пам'яті та фатального падіння усієї системи.

Рішенням може бути концепція “безпеки за замовчуванням”, реалізована в мові програмування Rust. На відміну від C/C++, Rust усуває цілі класи помилок на етапі компіляції завдяки моделі управління пам'яттю на основі правил власності та позичання. Принцип роботи цієї моделі, що унеможливлює використання пам'яті після її звільнення та забезпечує безпечне посилання на об'єкти, проілюстровано на рис. 1 [2].

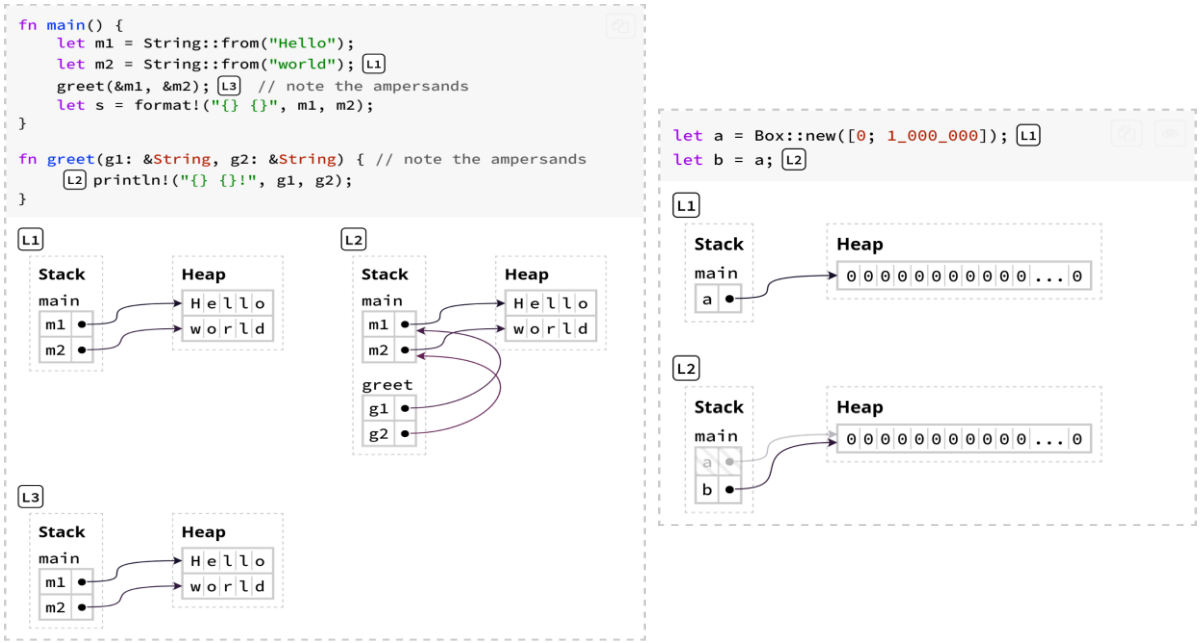


Рисунок 1 – Безпечне позичання (ліворуч)
та передача власності (праворуч) [2]

Замість використання ресурсомісткого збирача сміття, компілятор жорстко контролює життєвий цикл об'єктів. Крім того мова змушує явно обробляти позаштатні ситуації через типи Result та Option, виключаючи появу нульових вказівників, та ігнорування помилок. Прошивка не скомпілюється за наявності ризику виходу за межі буфера, доступу до вивільненої пам'яті чи ігноруванні помилок.

На практиці це радикально змінює наслідки логічних помилок. Спроба виходу за межі буфера в Rust спричинить контрольовану зупинку потоку виконання, а не перезапис сусідньої пам'яті. У поєднанні з апаратним сторожовим таймером це гарантує безпечне перезавантаження. Так критична вразливість знижується до рівня тимчасової відмови, повністю зберігаючи цілісність системи.

Ефективність підходу підтверджується кількісними показниками індустрії. За аналітикою розробників Google, близько 70% усіх критичних вразливостей у C/C++ проєктах є прямим наслідком помилок управління пам’яттю[3]. Використання моделі власності Rust превентивно усуває ці 70% загроз на етап збірки. Крім того, детермінована обробка помилок (через тип “Result” та оператор “?”) покращує архітектурні метрики коду.

Емпіричні дослідження показують, що код на Rust має об'єктивно нижчу цикломатичну складність V(G) порівняно з C/C++[4]. Це пояснюється тим, що у класичних програмах ручна перевірка вказівників створює десятки розгалужень (блоків “if”), експоненціально ускладнюючи граф потоку керування. У Rust автоматичне управління ресурсами дозволяє утримувати складність функцій на мінімальному рівні, значно знижуючи ймовірність логічних помилок.

Додатковою перевагою є безпека багатопотокової обробки. Завдяки системі маркерних типів “Send” та “Sync”, компілятор статично гарантує відсутність станів гонитви під час паралельного опитування датчиків та мережевих запитів, роблячи систему передбачуваною та стійкою до збоїв.

Цей підхід стає новим індустріальним стандартом. Наприклад, компанія-виробник чіпів серії ESP32 Espressif активно інвестує у розвиток інструментарію “esp-rs” як повноцінної та безпечної альтернативи своєму класичному C-фреймворку “esp-idf”. Офіційна підтримка Rust виробниками “заліза” свідчить про його стратегічну роль у побудові систем наступного покоління.

Отже, інтеграція Rust у розробку прошивок IoT є закономірним та обгрунтованим кроком. Усунення вразливостей пам'яті на етапі компіляції не лише запобігає кібератакам, але й радикально підвищує загальну відмовостійкість апаратних комплексів. Це зміщує фокус інженерів із тривалого низькорівневого налагодження на реалізацію надійної бізнес-логіки.

***Список використаних джерел:***

1. Fewer S. CVE-2026-2329: Critical Unauthenticated Stack Buffer Overflow in Grandstream GXP1600 VoIP Phones (FIXED). Rapid7. URL: https://www.rapid7.com/blog/post/ve-cve-2026-2329-critical-unauthenticated-stack-buffer-overflow-in-grandstream-gxp1600-voip-phones-fixed/ (дата звернення: 09.03.2026).

2. Crichton W., Krishnamurthi S., Nichols C. What is Ownership? The Rust Programming Language. URL: https://rust-book.cs.brown.edu/ch04-01-what-is-ownership.html (дата звернення: 09.03.2026).

3. Stoep J. V., Hines S. Rust in the Android platform. Google Online Security Blog. URL: https://security.googleblog.com/2021/04/rust-in-android-platform.html (дата звернення: 09.03.2026).

4. Evaluation of Rust code verbosity, understandability and complexity / L. Ardito та ін. PeerJ Computer Science. 2021. Т. 7. С. e406. DOI: https://doi.org/10.7717/peerj-cs.406

5. Старченко Д.М. Кібербезпека інтернету речей (IoT) / Д.М. Старченко // Радіоелектроніка та молодь у XXI столітті : матеріали 29-го Міжнар. молодіж. форуму, 16–19 квітня 2025 р. – Харків : ХНУРЕ, 2025. – Т. 3. – С. 535–537 (дата звернення: 09.03.2026).