專題研究計畫

112 科技部大專生研究計畫「以 Rust 程式語言改寫 Linux Kernel Module 以增強其記憶體管理安全性 - 以 ksmbd 模組為例」

一、摘要

本研究探討在 ksmbd [1] 中引入 Rust [2] 語言所帶來的影響，使用 Rust for Linux [3] 中 binding 等工具配合包裝函式 (wrapper function) 引入 Rust 語言，測試 Rust 語言與 C 語言之間不同實作的效能差距，並發現到 Rust 語言相比於 C 語言的實作有一定的效能減損，但使用 Rust 語言的實作具有編譯期與執行期安全性的優勢。

二、研究動機

Linux Kernel 5.15.61 版本在將 ksmbd (SMB3 Kernel Server) 合併到 Linux 核心主線 (mainline) 後，ksmbd 就被接連發現到許多與記憶體相關的安全性問題，如在 2022 年 12 月 24 日被發現到的因為 Use-After-Free 所導致的 RCE (Remote Code Execution) 漏洞，編號為 CVE-2022-47939 [4]，以及因為不當的參數檢查，造成 Heap Buffer Overflow 所導致 Kernel panic 的 CVE-2023-0210 [5]。在 Linux Kernel 6.1 版本中引入了 Rust 程式語言，Rust 語言為一門注重安全性以及效能的語言，特別是記憶體安全性的部分。雖然 Rust 語言在 Linux 核心中已經在驅動程式，如網路介面卡等領域進行了初步的嘗試，但是在一些安全性以及效率方面的驗證仍然有所改進，且在核心子系統中的應用較少相關的研究。因此，本研究將專注於 ksmbd 子系統，以探討引入 Rust 語言帶來的潛在好處同時，會有多少的效能損失以及需要面臨的挑戰。

三、研究背景

Rust 語言為一專為系統程式所設計的程式語言，在 Linux Kernel 6.1 版本中正式被引入，強調記憶體安全性以及高效能，為目前 Linux 核心未來發展的重要程式語言。目前對於 Rust 語言的使用以及研究大多聚焦於驅動程式中，如 Intel e1000 網卡 [6] 以及在 Linux Plumbers Conference 2022 由 Andreas Hindborg 所發表的 Linux Rust NVMe 驅動程式 [7]，思科工程師 Ariel Miculas 所發表的 PuzzleFS 驅動程式 [8] 等等，較少有對於核心子系統的研究。而 ksmbd 這個子系統在進入到 Linux 核心的主要分支後就被發現許多與記憶體安全有關的問題，如 Use After Free, Heap Overflow, Memory Leak 等等，我認為這是一個很好的研究目標，藉此機會，本研究嘗試探討在 ksmbd 中引入 Rust 語言所帶來的潛在好處以及需要面對以及解決的問題。

四、文獻回顧

在 Linux Device Drivers in Rust [9] 這一篇文獻中，使用 Rust 語言對 Out-of-tree 的驅動程式 Wireguard 進行重構，在重構的過程中作者使用了許多使用 Rust 語言包裝的 C 語言所撰寫的函式，而這也是目前 Rust for Linux 使用 Rust 語言對 Linux 核心模組 (Linux Kernel Module) 進行開發的作法，並提出了幾項關於 Rust 語言效率的說明，首先是 Rust 語言中許多抽象經過編譯器轉換成組合語言後顯示出的結果，並沒有產生出額外的開銷，作者稱為這是 Rust 語言的零成本抽象，間接證明 Rust 語言和 C 語言的執行效率是相當的，但我認為這是需要進行驗證的，特別是在動態記憶體分配時更是如此。而在 Linux Kernel Module Development with Rust [10] 中使用 Rust 語言對 /dev/null 和 /dev/urandom 這兩個設備進行重構，並展示使用 Rust 語言改寫的版本在讀寫性能上並不會有巨大的損失，但是在 Rust 語言與 C 語言的實作執行時間，效能方面沒有進一步的說明。

綜觀上面這兩篇文獻皆對 Rust 語言引入到 Linux 核心的開發或是驅動程式的開發皆為正向看法，在安全性方面有相比於 C 語言更好的抽象。而本研究將以 ksmbd 為例，改寫該子系統，並驗證使用 Rust 語言的實作相比於 C 語言的實作在使用 Rust for Linux 提供的封裝進行重構後，能夠帶來什麼安全性上的優勢，以及有多少的效能減損。

五、研究方法

本研究將基於 Rust for Linux 專案底下的 linux 專案中 rust 分支進行，之所以不使用目前 linux 主線分支，原因為截至目前 Linux Kernel 6.5.6 版本，對於 Rust 語言的引用還十分的保守，許多在 Rust for Linux 專案底下提供的封裝暫時尚未引入到 Linux 中，因此本研究將使用 rust 分支的 linux 進行。

對於安全性的部分，我們將針對 ksmbd 中 NTLMv2 驗證和 ksmbd 中 user 結構記憶體分配的部分進行改寫，並以 CVE-2023-0210 進行說明，從中看到 Rust 語言是如何運用其檢查機制在編譯期以及執行期保障系統安全，為了加入 Rust 語言的包裝函式，我們需要將從 C 語言傳入的型別以及結構等轉換成 Rust 語言中的型別，這部分參照 Rust for Linux 專案提倡的做法，藉由 binding 中 helper 與 bindings_helper 來產生出對應的巨集、型別和函式原型，概念如下圖 (一) 所示。

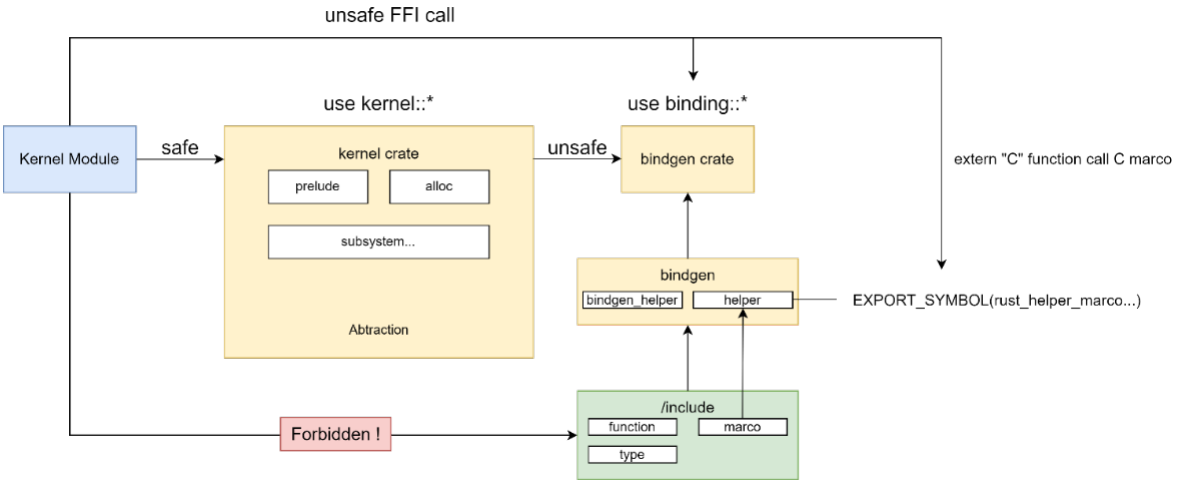


圖 (一) Rust for Linux 關於 C 語言與 Rust 語言相互綁定關係圖

C 語言中的函式會通過 Rust 語言撰寫的包裝函式去呼叫其他 C 語言撰寫的函式，這部分在 Rust 語言中將使用 unsafe 進行標記，而對於部分核心的 API，如在 C 語言中使用的 kmalloc 等函式，將會通過 Rust for Linux 中 rust 分支上已經完成的 alloc 抽象層進行封裝，如動態記憶體分配使用 Box 進行封裝，不直接去呼叫原始 C 語言提供用於記憶體操作的核心 API。由於經由 bindings_helper 所產生的函式原型在使用 EXPORT_SYMBOL_GPL 時暴露在外的符號會經過函式名稱修飾 (function name mangling) 用於編譯器分析，這會導致我們在使用一些工具，如 trace-cmd 檢視 function-graph 時追蹤不到 Rust 語言包裝的函式，因此在 C 語言中要呼叫到 Rust 語言所包裝的函式中間會再加上一層 C 語言所撰寫的函式抽象層，且這一層包裝層需要加上 noinline 的屬性，以利於分析呼叫堆疊以及行為。以下為實驗架構圖，圖 (二)。

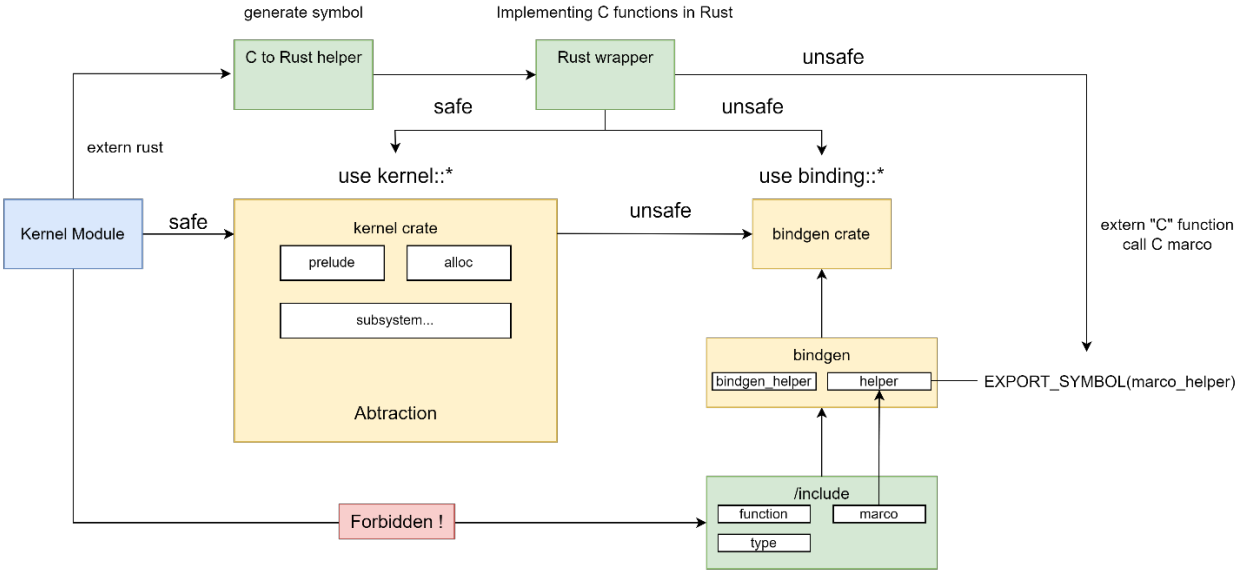


圖 (二) 實驗架構圖

六、研究成果

首先針對於安全性的部份，在 NTLMv2 原始的 C 語言實作中存在越界存取的問題，以 CVE-2023-0210 為例，在 ksmbd_decode_ntlmssp_auth_blob 函式中因為不當的邊界計算，接著呼叫 ksmbd_auth_ntlmv2 進行記憶體空間分配，分配完成後對該記憶體空間進行走訪以完成初始化操作，走訪的邊界為一個 size_t 的型別，也就是一個無號數，而在 kzalloc 為有號數，如果傳入的參數不當，將會導致分配的空間與初始化走訪的邊界不相等，導致在內核空間的越界存取，進而導致 Kernel panic。而在使用 Rust 語言的等效實作中，會在 ksmbd_decode_ntlmssp_auth_blob 函式計算完邊界接著呼叫 ksmbd_auth_ntlmv2 時發現到 underflow 的錯誤，底層是借助於 Rust 語言中型別系統，我們可以直接引用 Rust for Linux 的相關實作，在發現到 underflow 錯誤後，會直接走向 binding 中 BUG 巨集，而不會繼續 ksmbd_auth_ntlmv2 分配記憶體的操作，因此不會發生 Kernel panic。

對於上面的情況，可以推測出對於一個已經進入到 Linux 核心中的系統或是驅動程式，如果因為傳入的一些參數或是其他的錯誤發生時，對於 Rust 語言撰寫的子系統或是驅動程式，這種錯誤並不會導致 Kernel panic，因為 Rust 語言會試圖阻止這樣的操作，而對於 C 語言的實作則會因為沒有邊界確認等等機制，因此會直接發生 Kernel panic。由此看到 Rust 語言的實作相較於 C 語言的實作更具有安全性，但是這樣的安全性為在執行期間所保障的安全性，實際上這樣的程式碼在 Rust 語言中是能夠編譯成功的，Rust 語言主要的優勢為在編譯期保障其安全性，但是由於需要與核心 API 互動所需要的 FFI 呼叫，因此會犧牲掉在編譯期間進行的安全檢查。

接著是改寫記憶體分配的部分，使用 Rust 語言的抽象層對原始以 C 語言實作的 ksmbd_alloc_user 重新實作，在原始 C 語言版本中使用 kmalloc 進行記憶體分配，而對應到的 Rust 語言實作為使用 Box 進行分配，並將分配完成在記憶體堆積區域的指標回傳到 C 語言中用於後續的使用，在重構過程中利用 Rust 語言編譯期的檢查發現到原始 ksmbd_alloc_user 的實作缺失，沒有對結構中所有成員進行初始化，接著深入研究與分析，發現到 ksmbd_user 結構中 failed_login_count 成員已經不再被使用，Rust 語言要求在初始化階段需要對結構所有成員進行初始化，因此發現到此錯誤，目前已對該專案進行貢獻 [11] 並合併進入主線。從這個例子中可以看出 Rust 語言相比於 C 語言在編譯期間有更多的檢查確保程式安全性。

針對於效率的部份，我們測試在使用 Rust 語言對 ksmbd_decode_ntlmssp_auth_blob 進行改寫的情況下，在執行 100 次測試並取其平均值的情況下，Rust 語言相比於 C 語

言的執行速度大約慢了 600 到 700 微秒，和 C 語言有 36% 的效能差距，而在 ksmbd_alloc_user 的測試中，Rust 語言版本相比於 C 語言的版本執行時間幾乎相等，慢了 10 到 15 微秒，大約是 4% 的效能差距，統計圖表如圖 (三)，圖 (四) 所表示。

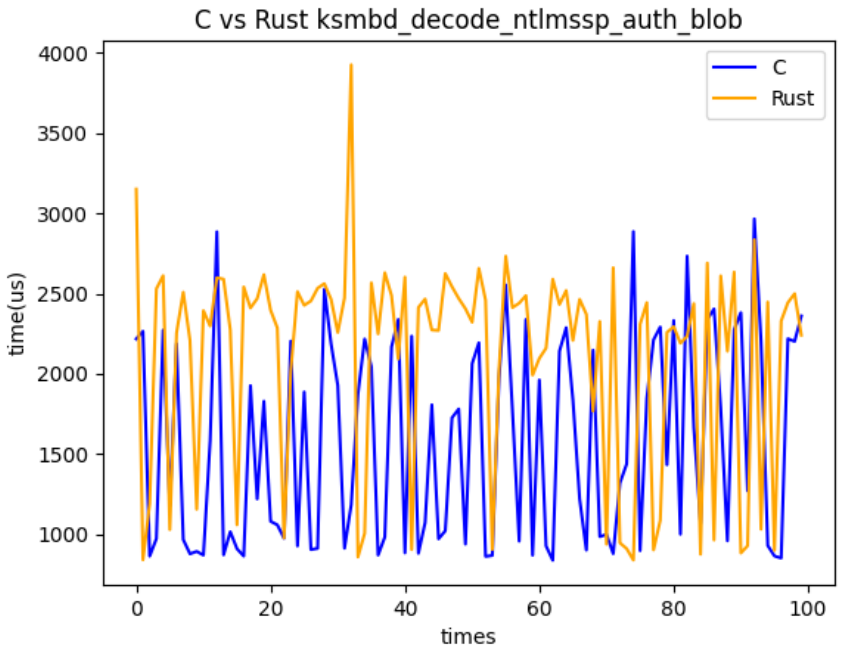
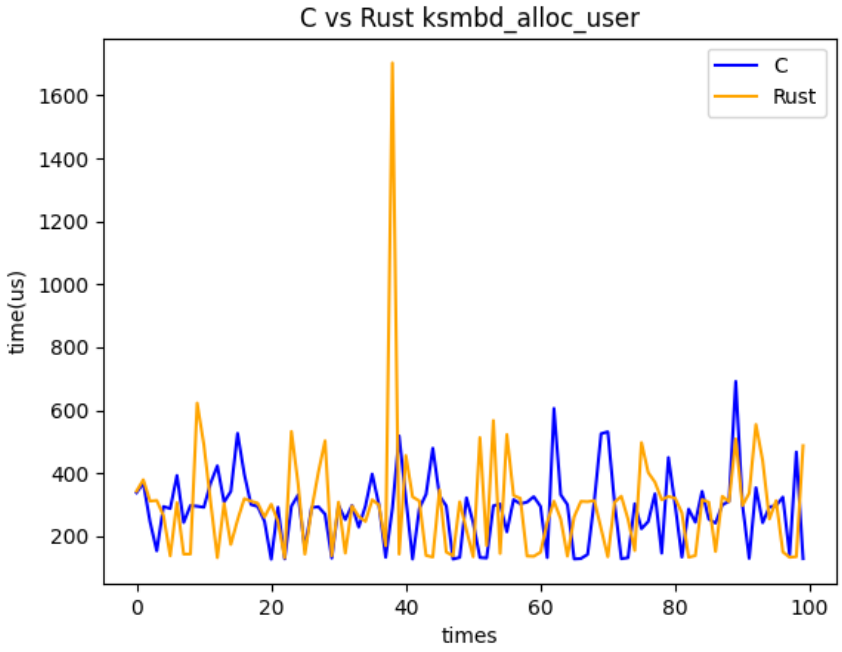


圖 (三) 使用 C 語言與 Rust 語言實作 ksmbd_decode_ntlmssp_auth_blob 執行 100 次下執行時間統計



圖（四）使用 C 語言與 Rust 語言實作 NTLMV2 協定中 ksmbd_alloc_user 執行 100 次下執行時間統計

Rust 語言與 C 語言之間效能差距可以歸納於以下幾點。

1. 使用 FFI 呼叫 Linux 核心提供的 API 所需要的成本
2. 在 Rust 語言中要使用 C 語言的巨集，目前作法為使用 helper.c 將巨集使用函式進行封裝，接著通過 EXPORT_SYMBOL 讓 Rust 語言能夠使用 FFI 進行呼叫，如此多出了函式呼叫成本與 FFI 呼叫的成本
3. 對於 inline 函式，同樣也是需要先用 C 語言進行封裝，接著 Rust 語言通過 FFI 進行呼叫
4. 對於 Rust 語言撰寫的程式碼如果要進行測試，由於函式名稱修飾 (function name mangling) 的關係，需要加上一層 C 語言撰寫的包裝函式進行包裝以使用 Linux 核心中提供的分析工具，如 ftrace 等工具進行分析

總結來說，Rust 語言撰寫的驅動程式或是核心模組確實相比於 C 語言實作提供了更多的安全性檢查機制，但效能部份，還有許多需要優化的部份，如 C 語言的巨集部份目前只有部份以 Rust 語言的巨集進行重構，舉例像是位於 rust/binding 中 lib.rs 的 container_of，大部分巨集仍然需要借助於 helper.c 將其轉換為函式呼叫以提供給 Rust 使用，這部份還有很大的優化空間，而對於產生 C 語言對應的 Rust 語言呼叫界面，目前 bindings_helper.h 還有許多問題，如針對 asm/byteorder.h 的處理會有許多的問題，目前這一些工具皆處於實驗階段。

Rust 語言確實為目前系統開發的趨勢，相比於 C 有更好的安全性，以及更加優秀的抽象類別，有利於程式碼的閱讀以及開發，但是目前還有許多工作需要開發人員的參與。

七、參考文獻

1. "KSMBD - SMB3 Kernel Server" https://docs.kernel.org/next/filesystems/cifs/ksmbd.html
2. "Rust lang" https://www.rust-lang.org/zh-TW
3. "Rust-for-Linux." https://github.com/Rust-for-Linux/linux
4. "CVE-2022-47939" https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-47939
5. "CVE-2023-0210" https://nvd.nist.gov/vuln/detail/CVE-2023-0210
6. "fujita/rust-e1000" https://github.com/fujita/rust-e1000
7. "Andreas Hindborg Linux (PCI) NVMe driver in Rust LPC'22"
8. "puzzlefs" https://github.com/project-machine/puzzlefs
9. "Zach Schuermann, Kundan Guha Linux Device Drivers in Rust" https://zachschuermann.com/static/6118.pdf
10. "Shao-Fu Chen, Yu-Sung Wu Linux Kernel Module Development with Rust" https://ieeexplore.ieee.org/document/9888822
11. "ksmbd: remove unused field in ksmbd_user struct" https://lore.kernel.org/all/20231002053203.17711-1-hank20010209@gmail.com/T/#u