Toward building memory-safe network functions with modest performance overhead

Keunhong Lee*, Shinae Woo*, Sanghyeon Seo†, Jihyeok Park*, Sukyoung Ryu*, Sue Moon*
KAIST* Hyperconnect, Inc.†

1. INTRODUCTION

For the past few decades network functions have evolved much in variety and complexity. Contributing factors behind are ever-increasing link speeds, novel hardware features, and new network designs (e.g., datacenters), just to name a few [4, 6]. In order to deliver high performance in network I/O, network functions rely heavily on low-level memory management. Yet, such memory management is error-prone. Among the Linux CVEs (Common Vulnerabilities and Exposures) for network functionality with CVSS(Common Vulnerability Scoring System) score 7.5 or higher, more than 30% are due to memory management faults: buffer overflow, null pointer dereferences, use-after-free, and double free. Table 1 shows some examples of critical vulnerabilities from memory-related bugs on the kernel's network stack. Such bugs often lead system crash, systems being compromised, or leakage of secret information.

In traditional type-safe languages (e.g., ML, Haskell, Scala), memory-safety is preserved by disallowing pointer manipulation; they automatically manage memory resources without explicit `malloc()/free()`, restrict accesses beyond memory bound, and have no null pointer dereferences. There are notable projects of implementing network function using type-safe languages [5, 2, 1]. Yet for performance-critical network functions, intermittent execution of a garbage collector incurs unbearable performance penalty.

Rust [3], a recently developed language for safe system programming, aims to support memory-safety without garbage collection via its advanced type system. However, the type system restricts programmability; even frequently used data structure implementations such as doubly linked lists cannot satisfy the type system.

In this work, we report that network functions' architectures have specific features that suit well with the Rust type system. We illustrate how the Rust type system supports real-world network functionality intuitively using concrete code examples. Our prototype implementation also shows that the Rust implementation has comparable run-time performance to a corresponding C implementation.

2. RUST: ZERO-COST MEMORY-SAFETY

Rust's [3] type system guarantees memory-safety without run-time overhead. Rust developers manipulate low-level memory without garbage collection as in C/C++, while the strongly-typed memory segment and the *ownership* abstraction enables compile-time verification of memory-safety. In Rust, only a single variable has an ownership of a memory segment, which allows an exclusive update access to it. The ownership *linearly moves* among variables and multiple *immutable borrowing references* (read-only) for the memory segment enables high concurrency whenever possible.

This restricted programming model makes certain data structures hard to implement. For example, every element of a doubly-linked list requires at least two writable references and does not fit Rust's ownership. Therefore, in addition to the default type system, Rust developers need to add unsafe code blocks (*e.g.*, calling native C functions, doing pointer arithmetic) or low-performance abstractions such as reference counting wrappers `Rc<RefCell>>` requiring C-like pointer semantics.

We compare the performance of Rust and C implementations of an IPv4 reassembler. It is a simple network function, yet only with per-flow state management. The Rust implementation shows only 17% performance degradation in throughput, while it guarantees memory safety. This performance gap may shrink if we optimize common data structures (*e.g.*, hash table) like C.

3. NETWORK FUNCTION ARCHITECTURE

We argue that the core architecture of network functions fits well with Rust's type system such that network functions should reap the benefit of zero-cost abstraction for memory safety as shown in Figure 1.

Rust-friendly RW patterns Network functions manage a 5-tuple context of network flows. To exploit the parallelism in the multicore architecture, input packets are evenly distributed to multiple cores preserving flow affinity. Then, network function's data can be classified into two categories: *Per-flow data* accessed within a single network flow, and *global data* accessed across different network flows. The per-flow data is frequently read and updated, but exclusively within a single thread. The ownership of per-flow data is assigned when its flow context is assigned to a thread. The global data is shared among multiple threads. Since it is mostly read and rarely updated, there is enough possibility of parallelism by having read-only references. Such use of sin-

CVE item	CVSS Score	Function	Kernel version	Description
CVE-2014-2523	10	NetFilter	3.13.6	Incorrect use of pointers which allows remotely crash system or execute arbitrary code
CVE-2015-1421	10	SCTP	$< 3.28.8$	Use-after-free which allows remote DoS attack or possible other impacts
CVE-2015-8787	10	NetFilter	< 4.4	NULL pointer dereference which allows remotely crash system
CVE-2016-7117	10	Socket	$< 4.5.2$	User-after-free which allows remotely execute arbitrary code
CVE-2016-9555	10	SCTP	$< 4.8.8$	No memory bound check which allows remote DoS or possible other impacts

Table 1: Examples of critical CVEs of memory-related bugs on network stack.

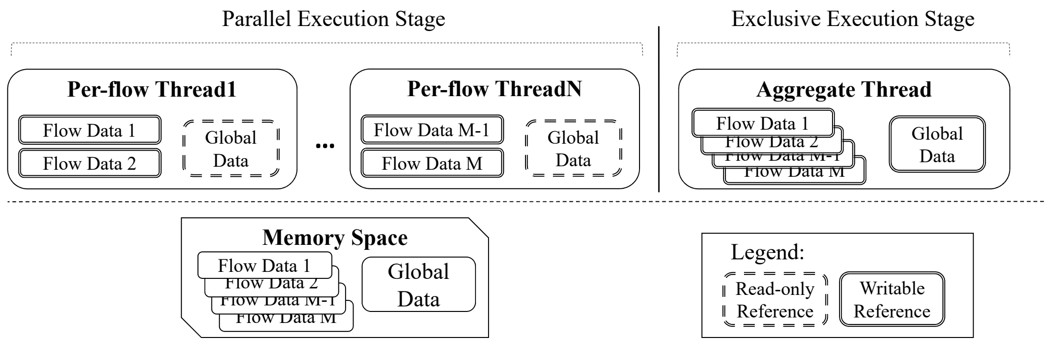

Figure 1: Network Function Architecture exploiting Rust's type system

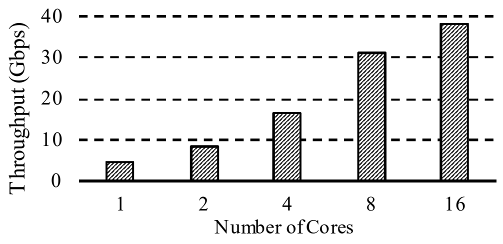

Figure 2: Performance of NAT on Rust

gle ownership without reference counting delivers Rust's memory-safety with low overhead to network functions.

4. EXPERIENCES OF IMPLEMENTING NAT USING RUST

We implement NAT (Network Address Translator) using Rust. NAT has a address mapping table that is globally shared among threads. For every packet arrival, the table is accessed for read, and updated when there is a new outgoing flow. Our NAT implementation consists of two types of packet handlers: one that reads the table and performs the address translation (Per-flow threads in Figure 1), and the other that updates the table when there is a new network flow (Aggregate thread in Figure 1).

Each per-flow thread access the global data including the address lookup table as a read-only reference, and the per-flow data as a writable reference, resulting in exclusive access. The aggregate thread requires a writable reference to both global data and per-flow data to perform the lookup table update. Thus, when the need for aggregation arises, the parallelism collapses and only a single aggregation thread accesses the global data. However, this aggregation happens only when a new flow is created and the per-flow handlers can run in parallel most of the time.

We evaluated our NAT implementation on 24-core Xeon E5-2670v3 machine connected with two Mellanox ConnextX-3 40G NICs. One NIC is used for the internal network and the other for the external network. The input workload is 100k UDP flows with 512B packets. Figure 2 shows the performance of our implementation with varing numbers of CPU cores. Our implementation scales well when the number of core increases and almost saturates the 40G link with 16 cores.

5. FUTURE WORK

In this work we have built network functions in Rust, with type-safe guarantees and modest performance overhead. We plan to investigate other characteristics of network functions – layered architectures, (de-) multiplexing of data flows – and seek additional optimization opportunities in Rust's type system.

Acknowledgement

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2017-0-01731,Type-safe networking stack for IoT devices).

6. REFERENCES

[1] go-tcpip. `https://github.com/unigornel/go-tcpip`.
[2] HaNS. `https://github.com/GaloisInc/HaNS`.
[3] Rust. `https://www.rust-lang.org`.
[4] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *ACM SIGCOMM CCR*, 2010.
[5] E. Biagioni, R. Harper, and P. Lee. A network protocol stack in standard ml. *Higher-Order and Symbolic Computation*, 2001.
[6] R. Mittal, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats, et al. TIMELY: RTT-based Congestion Control for the Datacenter. In *ACM SIGCOMM CCR*, 2015.