Rusty ZMap: Fearlessly Fast Internet-Wide Scanning

Olayinka Adekola
Stanford University

Kamran Ahmed
Stanford University

Tristen Nollman
Stanford University

ABSTRACT

ZMap is an Internet network scanning tool, originally implemented in C. We have re-implemented ZMap in Rust. This paper presents the design and implementation of Rusty ZMap, highlighting the challenges and solutions that were encountered while writing our Rust implementation, ZMap-rs. ZMap-rs not only retains the performance characteristics of the original ZMap but also provides enhanced safety and maintainability. We discuss the implications of our design choices and provide a comparative analysis based on our results alongside the original ZMap results.

1 INTRODUCTION

Network scanning is a fundamental technique used in network security and research, enabling the discovery of active hosts, services, and vulnerabilities across large address spaces. ZMap [2] revolutionized this field by providing a high-speed, efficient network scanner capable of probing the entire IPv4 address space for a single open port in under an hour. Despite its performance, ZMap's implementation in C poses significant challenges related to safety and concurrency, leading to poor maintainability and high complexity.

In this work, we introduce ZMap-rs, a re-implementation of ZMap in Rust, a modern systems programming language designed to prioritize memory safety, concurrency, and performance. Rust's ownership model and borrow checker enforce strict memory safety guarantees at compile time, preventing common issues such as dereferencing null pointers, dangling pointers, and data races. Additionally, Rust's robust type system and concurrency features provide significant advantages over the manual memory management and concurrency control mechanisms inherent in C.

Our implementation utilizes the broader Rust ecosystem of libraries (crates), such as the `etherparse` crate which allowed us to handle the serialization of Ethernet packets without using unsafe Rust. The re-implementation of ZMap allows us to evaluate the feasibility and benefits of writing systems software in Rust, particularly in terms of safety, maintainability, and performance. In Section 2.2, we talk more about the crate ecosystem in Rust and how it was able to help us with challenges we faced while writing ZMap-rs, like serialization and checksum calculation.

Finally, we present a comparison of performance metrics between the original ZMap and ZMap-rs, demonstrating that our implementation maintains the high-speed scanning capabilities of the original while providing enhanced safety and maintainability. Through ZMap-rs, we aim to highlight Rust's potential as a strong language for developing **high-performance, reliable, and maintainable** systems software.

2 BACKGROUND

Modern Internet scanning is a fundamental practice in the field of cybersecurity and network management. It involves systematically probing a large number of IP addresses to identify active hosts, running services, potential vulnerabilities, or deployment of patches. This process is crucial for maintaining robust security postures, as it allows organizations to detect and mitigate threats before malicious actors can exploit them. Additionally, Internet scanning is vital for academic research, providing insights into Internet topology, traffic patterns, and the overall health of the global network infrastructure.

2.1 Network Scanning

Nmap: The Versatile Network Scanner. Nmap is one of the most widely used tools for network scanning and auditing. It allows users to discover devices and services on a network, perform security audits, and monitor network uptime. Nmap operates by sending various types of packets to specified IP addresses and analyzing the responses. It supports a range of scanning techniques, including TCP SYN scans and UDP scans, among others. Nmap's flexibility and extensive scripting capabilities (through the Nmap Scripting Engine) enable it to perform detailed scans that can identify open ports, operating systems, and service versions, providing a comprehensive overview of a network's security status.

ZMap: High-Speed Internet Scanning. While tools like Nmap are highly effective for detailed network scans, they may not be suitable for scanning the entire Internet due to speed constraints. This is where ZMap comes into play. ZMap is designed for high-speed Internet scanning of a single port, capable of probing the entire IPv4 address space in a matter of ~45 minutes. Developed by researchers at the University of Michigan, ZMap achieves this remarkable speed by sending probe packets at a high rate and efficiently analyzing the responses. The tool uses raw sockets to bypass traditional networking stacks, allowing it to send and receive packets with minimal kernel overhead.

Network scanning with ZMap involves sending probe packets (such as TCP SYN, ICMP echo requests, or application-specific queries) to a vast number of IP addresses and recording the responses. This process helps in discovering active hosts, services, and potential vulnerabilities across a broad network landscape. ZMap's high-speed capabilities make it particularly useful for large-scale network security audits, academic research, and real-time monitoring of global network events.

The importance of ZMap and similar tools lies in their ability to provide a macro-level view of the Internet. They facilitate the identification of widespread vulnerabilities, the assessment of Internet-wide security practices, and the collection of data essential for understanding and improving network resilience. For example, ZMap has been used in various research projects to measure the prevalence of security protocols, track the adoption of encryption standards, and analyze the distribution of specific types of network infrastructure [2, 3].

Modern Internet scanning is a critical practice for ensuring network security and understanding the global Internet landscape. Tools like Nmap and ZMap each play a unique role in this process.

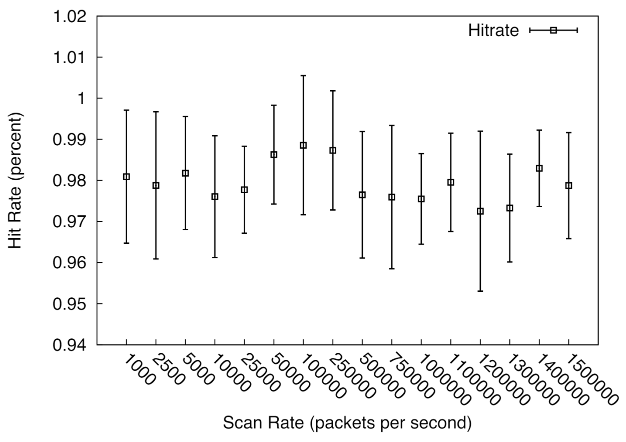

Figure 1: Reproduction of Figure 2 from ZMap. The ZMap authors find no statistically significant correlation between hit rate and scan rate suggesting that ZMap can handle scanning at upwards of 1.4 million packets per second. The mean and standard deviation over ten trials are shown for each scan rate.

Nmap excels in detailed, flexible scans suitable for in-depth security audits of smaller networks, while ZMap provides the capability to quickly scan the entire Internet, making it invaluable for large-scale assessments and research. Together, these tools contribute to a more secure and well-understood Internet by enabling the discovery and mitigation of vulnerabilities, the analysis of network behavior, and the advancement of cybersecurity research.

The primary performance benchmarks of ZMap revolve around its capability to scan the entire IPv4 address in under 45 minutes, coupled with an overall hit rate of around 0.98%. Hit rate, in this context, signifies the proportion of successfully responded packets over the total number of probes sent. Given these metrics, our focus centered on reproducing Figure 1 and scanning the IPv4 space in under 45 minutes.

2.2 ZMap

As aforementioned, ZMap is an open-source tool designed for large-scale studies of hosts and services on the public Internet, making it an essential tool for information security research and vulnerability detection. Here, we highlight ZMap's key properties:

- **Fast scanning.** ZMap is capable of scanning the entire IPv4 address space in around 45 minutes on a 1 Gbit/s network connection, reaching ~98% theoretical line speed.
- **Efficient IP address iteration.** ZMap uses a novel approach to scanning, employing cyclic multiplicative groups to iterate through the address space, allowing it to scan the same space roughly 1,300 times faster than Nmap.
- **Probe modules.** ZMap includes various probe modules for different protocols that are responsible for constructing probe packets and validating responses. The original implementation includes probe modules for TCP SYN and ICMP echo scanning.
- **Blocklist.** To support responsible scanning, ZMap includes a blocklist feature that allows users to specify IP address ranges to exclude from scans. This feature helps prevent the scanning of IANA-reserved addresses and other ranges that do not wish to be scanned.
- **Reproducibility.** ZMap supports using a seed to deterministically pick a permutation of the address space. This feature ensures that the tool scans addresses in the same sequence across multiple runs, ensuring reproducibility.

ZMap is a powerful tool for large-scale network scanning and vulnerability detection. Its speed and scalability make it an essential tool for information security research and professionals. However, its resource-intensive nature and complexity may limit its use in certain environments that lack the necessary networking resources. We present the following critiques of the original ZMap implementation:

- **Interpretability.** One significant critique of ZMap is the interpretability of its code. The codebase can be challenging to understand, primarily due to its reliance on idiomatic C constructs such as type casting, type punning, and raw pointer arithmetic. These practices, while efficient, often obscure the purpose of the code and make it difficult to follow at a low level. Attempting to translate the code line-by-line to Rust was not effective. It wasn't immediately apparent how the different components interacted with each other, and more context was needed to understand how they all fit together.
- **Data race.** Another issue with the original ZMap is the presence of a data race in its global state, specifically related to the number of sent packets that is being modified by sender threads and read by a monitor thread. Although this data race has not caused problems in our observations, it represents an area for improvement. Data races can lead to unpredictable behavior and are generally undesirable. To address this, in our implementation of ZMap, we ensured that any shared state accessed by multiple threads is protected by mutexes and wrapped in atomic references.

2.3 Rust

Rust is a modern systems programming language that has garnered significant attention due to its focus on performance, safety, and concurrency. Unlike traditional languages such as C, Rust guarantees memory safety without the need for a garbage collector. This is achieved through a combination of compile-time checks and runtime validations, making Rust particularly well-suited for systems programming. We outline some of Rust's key features below.

Memory safety. A cornerstone of Rust's safety guarantees is its borrow checker, a compile-time system that ensures all references point to valid memory. This mechanism prevents a range of common programming errors, including null pointer de-references, dangling pointers, and data races. By enforcing strict rules about ownership and borrowing, Rust eliminates many of the vulnerabilities that plague C programs, providing a more robust foundation for software development.

Compile-time checks. Rust's compiler performs extensive checks at compile-time to ensure memory and type safety. These checks include verifying that all references are valid and that types are used correctly, preventing many runtime errors. The compiler's rigorous analysis helps developers catch bugs early in the development process, leading to more reliable and maintainable code.

Performance. Performance is another key advantage of Rust. By compiling to LLVM intermediate representation (IR) and machine code, Rust generates highly optimized binaries comparable to those produced by C and C++ when compared to languages with similar levels of memory safety like Java or Go. Rust's lack of a garbage collector ensures predictable performance, which is crucial for real-time and high-performance applications.

Type safety. Rust's type system is designed to enforce type safety at runtime, preventing a wide range of errors related to type mismatches and invalid type casts. This strong emphasis on type safety helps developers write robust code, as the compiler enforces correct type usage and prevents many common programming errors.

Ownership and borrowing. Central to Rust's design are the concepts of ownership and borrowing. Ownership in Rust means that each value has a unique owner responsible for its memory management. Borrowing allows temporary access to a value without transferring ownership. Rust enforces strict rules around borrowing, ensuring that references are valid and preventing issues such as use-after-free errors.

Concurrency. Rust's design makes it particularly well-suited for concurrent and high-performance applications. The language provides powerful abstractions for concurrent programming, such as channels for message passing and the `tokio` runtime for asynchronous programming. These features enable developers to write efficient, concurrent code that leverages modern multi-core processors, making Rust an excellent choice for developing high-performance systems software.

In summary, Rust's focus on memory safety, type safety, and concurrency provides significant advantages for systems programming. Its robust safety guarantees, combined with its performance and concurrency capabilities, make Rust an attractive choice for developers seeking to build reliable and efficient software. By leveraging these features, we can develop tools like ZMap-rs that maintain high performance while offering enhanced safety and maintainability.

3 DESIGN AND IMPLEMENTATION

In this section, we discuss the design and implementation of ZMap-rs, focusing on the key components of the original ZMap tool and how we adapted them to Rust. We also discuss the challenges we faced during the implementation and how we overcame them.

3.1 Packet Serialization

An overview of potential serialization strategies is discussed below and in Figure 2. We first discuss the challenges of serialization in Rust, followed by our initial approach using the `etherparse` library, and finally, our optimized approach using a pre-computed buffer and Rust's `copy_from_slice` method which doesn't require unsafe Rust, but is still similar to ZMap's serialization strategy.

Serialization in Rust. Serialization in Rust can be challenging due to its stringent memory management and ownership rules enforced by the borrow checker. The borrow checker ensures that all references point to valid memory, preventing common issues like dereferencing a null pointer and data races.

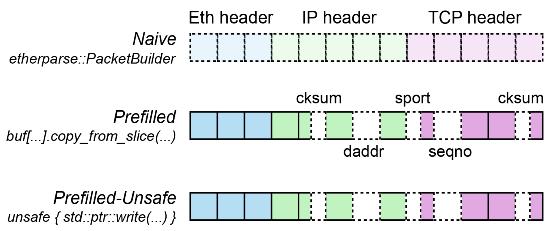
Eth header
IP header
TCP header
Naive
etherparse::PacketBuilder
cksum
sport
cksum
Prefilled
buf[...].copy_from_slice(...)
daddr
seqno
Prefilled-Unsafe
unsafe { std::ptr::write(...) }

Figure 2: Serialization strategies. A naive approach uses `etherparse::PacketBuilder` to construct a packet every time. However, since the majority of Ethernet, IP, and TCP header fields are known upon initialization, it's possible to cache the packet and only update the remaining fields at runtime. A C-like approach directly copies data like the IP destination address and checksum, along with the TCP source port, sequence number, and checksum using unsafe operations. This method avoids the need for bounds checking, whereas an alternative method using safe Rust (`copy_from_slice`) does incur this penalty. However, we didn't observe a significant performance impact when using this safe method.

However, this strict system can complicate tasks like packet serialization where mutable access to particular struct members and data transformations are required. This becomes a problem when these access patterns do not fit neatly within Rust's ownership model and type system, since it prevents the borrow-checker from being able to verify the correctness of the program. A trivial example of this is when ZMap uses C type casts to help serialize Ethernet packets (Listing 1). This pattern is not allowed in Rust unless there is a defined way of transforming the source type into the destination type, usually via a pattern like `try_into` or resorting to unsafe Rust (Listing 2). Another common pattern in ZMap involved passing packets around via pointers. However, this approach isn't feasible in Rust because direct manipulation of pointers is not allowed without resorting to unsafe Rust, which is discouraged. Instead, we had to abide by Rust's ownership model, which brought unique challenges when thinking of how to efficiently serialize our Ethernet packets, which was likely not something that needed a lot of thought for the original C implementation.

We had to handle transforming our Rust Ethernet packets into line-transmissible data, which is inherently unsafe. We aimed to handle these challenges without resorting to unsafe Rust since it required a lot of care when writing our implementation. We also wanted to see if we could match ZMap's performance without writing any unsafe code.

Packet handling with `etherparse`. To overcome some of the serialization challenges, we first used the `etherparse` library, which provides a set of utilities for parsing and building network packets. `etherparse` helps manage the low-level details of packet construction, ensuring that headers and payloads are correctly formatted and checksums are properly calculated. This allowed us to avoid calling `unsafe` ourselves while still handling packet serialization. This approach turned out to be inefficient since we had to build the entire packet in memory before sending it every time. Ideally, we would like to avoid this overhead by keeping as much of the packet

```c
int make_packet(void *buf, ipaddr_n_t src_ip,
                ipaddr_n_t dst_ip, uint32_t *validation,
                int probe_num)
{
  struct ethhdr *eth_header = (struct ethhdr *)buf;
  struct iphdr *ip_header
      = (struct iphdr *)(&eth_header[1]);
  struct tcphdr *tcp_header
      = (struct tcphdr *)(&ip_header[1]);

  uint16_t src_port
      = zconf.source_port_first
        + ((validation[1] + probe_num) % num_ports);

  ip_header->saddr = src_ip;
  ip_header->daddr = dst_ip;

  tcp_header->source = htons(src_port);
  tcp_header->seq = validation[0];
  tcp_header->check = 0;
  tcp_header->check = tcp_checksum(
      sizeof(struct tcphdr), ip_header->saddr,
      ip_header->daddr, tcp_header);

  ip_header->check = 0;
  ip_header->check
      = ip_checksum((unsigned short *)ip_header);

  return EXIT_SUCCESS;
}
```

Listing 1: ZMap casts a packet buffer, buf, to various networking-related structs to update the packet's fields. This is a common technique in C, but is not idiomatic nor allowed in safe Rust.

buffered as possible and only updating the fields that change. ZMap performs this "caching" to prevent extra kernel overheads, such as routing table and ARP network lookups, which hinder performance. While `etherparse`'s utilities helped us get a working, yet naive serialization implementation using safe Rust, it was too slow to match the C implementation of ZMap.

Ultimately, we chose a different approach which allowed us to cache as much of the Ethernet packet as possible and only update the fields that were dependent on the probe being sent (e.g., IP destination address and TCP source port). We were able to do this using slices and Rust's `copy_from_slice` method. This allowed us to cache much of the Ethernet packet the same way that ZMap did. However, we did so without the explicit use of pointers or type casts, which were heavily used in the original ZMap. Instead, we were able to strictly use safe Rust to achieve this same technique. This was possible since we knew the layout of our data was fixed. For example, we know where the TCP sequence number resides in our packet, so we can access those bytes directly via a slice in Rust. This allowed us to keep many of the fields that were not changing from packet-to-packet, while still mutating a small portion of the IP and TCP headers we needed for each probe. All of this worked together to allow efficient packet serialization in a very similar way as the original ZMap implementation, but with safer access patterns.

```rust
fn make_packet(
  &mut self,
  destination_ip: &Ipv4Addr,
  validation: &[u32],
  probe_num: u32,
) -> &[u8] {
  unsafe {
    let ip_header =
      &mut *(self.buffer.as_mut_ptr().add(ETH_HDR_SIZE)
        as *mut iphdr);
    ip_header.daddr = u32::from(*destination_ip).to_be();
    let ip_checksum = ip_checksum(
      &self.buffer
        [ETH_HDR_SIZE..ETH_HDR_SIZE + IP_HDR_SIZE],
    );
    ip_header.checksum = ip_checksum.to_be();
  };

  let num_ports = (self.source_port_last
    - self.source_port_first + 1) as u32;
  let src_port = self.source_port_first
    + ((validation[1] + probe_num) % num_ports) as u16;

  unsafe {
    let tcp_header = &mut *(self
      .buffer
      .as_mut_ptr()
      .add(ETH_HDR_SIZE + IP_HDR_SIZE)
      as *mut tcphdr);
    tcp_header.source = src_port.to_be();
    tcp_header.seq = validation[0].to_be();
    let tcp_checksum = tcp_checksum(
      &self.buffer[ETH_HDR_SIZE + IP_HDR_SIZE..],
      TCP_HDR_SIZE as u16,
      self.source_ip.into(),
      (*destination_ip).into(),
    );
    tcp_header.checksum = tcp_checksum.to_be();
  };
}
```

Listing 2: Here is a direct conversion of the type casting seen in Listing 1 to Rust in our `UnsafeProbeGenerator` struct. Note the use of `unsafe` throughout the code to handle type casting and raw pointer manipulation which was tedious and difficult to implement correctly. We wrote this code as an exercise and did not use it for any experiments.

Rust's ownership model and borrow-checker allowed us to write our implementation without making any explicit memory management calls in our code. Instead, the compiler handles it all at compile time. This is radically different from the original ZMap, which had to manually manage all of its memory. While memory management was not highlighted in the ZMap paper, not having to deal with explicit management was a significant mental weight lifted from our shoulders when we undertook this re-implementation.

Overall, it seems that serialization was easier in the C implementation while managing memory and per-thread state was more difficult. On the other hand, with Rust, serialization proved more complex, but managing memory and per-thread state was much simpler. The only challenge with Rust was handling global state across

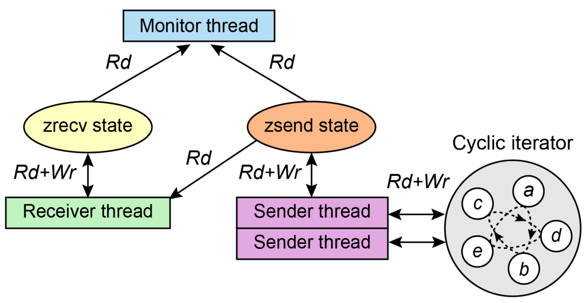

Figure 3: Shared state in ZMap. State is shared between multiple threads of the system to relay information about the current state of the scan to the monitor thread. The monitor thread displays scan progress and statistics to the user. Sender threads also share send statistics such as the number of packets sent. They also need to share a cyclic iterator to determine which IP address to send to next. Zippier ZMap [1] uses a similar design but shards the IP address space so that sender threads don't have to communicate as much and share the same iterator. Ultimately, ZMap's design requires a lot of shared state, which can be difficult to manage but is explicitly typed in Rust. This allows for the borrow checker to ensure that shared state is accessed safely.

multiple threads even though Rust makes thread-safe primitives explicit through the type system. It was harder to make concurrency bugs, but it was also hard to cooperate with the borrow checker.

3.2 Concurrency

ZMap maintains a large amount of global state, which is shared between monitor, sender, and receiver threads (Figure 3). As aforementioned in our ZMap critique, some of this state is mutable and exhibits data races. ZMap's monitor thread reads the global state to print statistics, while the sender thread writes to the global state to track the number of packets sent. We presume this is a benign data race and only results in the monitor thread reading stale data.

Rust embraces shared-state concurrency and its ownership model prevents these accidental races from occurring in the first place. This model allows safe concurrent access to shared resources and forces the programmer to *explicitly* handle shared mutable state. In our port of ZMap, we had to refactor the global state to be thread-safe. We achieve this by wrapping the sender and receiver shared state in an `Arc<Mutex<T>>` to ensure thread safety (Listing 3). `Arc` is a reference-counted pointer that allows multiple threads to share ownership of the same data while `Mutex` provides interior mutability much like Rust's `Cell` types, allowing the data to be mutated even when shared between threads. The `Mutex` type ensures that only one thread can access the data at a time, preventing multiple threads from mutating the data concurrently at runtime.

We thought this was a good example that illustrates how Rust's ownership model can be used to ensure thread safety and can raise compile time errors when the programmer tries to access shared data without proper synchronization. Additionally, the fact that a type is safe to share between threads is explicit in its signature makes it easier to reason about the code. Some of us argued that the Rust mutex API is less ergonomic than traditional synchronization

```
pub struct Context {
  pub config: Config,
  pub validate_ctx: AesCtx,
  pub sender_state: Arc<Mutex<SenderState>>,
  pub receiver_state: Arc<Mutex<ReceiverState>>,
}
```

Listing 3: Context object used by threads in ZMap-rs. It contains the CLI configuration, AES context for packet validation, and sender and receiver state. Most importantly, the sender and receiver state are shared between threads and are protected by a mutex.

primitives and that having a regular lock interface without any associated data would be more convenient. However, after porting ZMap, we found that the Rust mutex API was not significantly more difficult to use than `pthread`'s API. This also allowed us to clearly express what data was shared between threads as the lock and data itself were co-located. As observed in the original implementation, having them as separate objects can lead to concurrency bugs where the lock is forgotten or not held when it should be.

We were a bit concerned that the overhead of this synchronization would be significant and would degrade performance. We were considering alternative strategies to maintain performance such as using message passing between threads or updating the global state less frequently by sharding state. However, we found that the performance of ZMap-rs was comparable to the original C implementation as we demonstrate in our results. This may become a problem if we scale ZMap-rs to use more threads or scan at a higher rate than 1 Gbit/s, but we did not encounter this issue in our experiments and leave this to future work.

3.3 Networking

A fundamental aspect of ZMap involves transmitting raw Ethernet frames to reduce unnecessary kernel overhead. This is achieved by using raw sockets to send data directly to the network interface without any additional processing by the operating system. In the original ZMap implementation, this was done using the `socket` and `sendto` functions in C, which required the use of low-level network programming techniques.

In our Rust implementation, we encapsulated this functionality in a `RawEthSocket` struct, which provides a safe interface for sending raw Ethernet frames (Listing 4). The use of unsafe code is limited to the `sendto` function call, which is necessary to interact with the low-level network programming APIs provided by the operating system. This approach allows us to maintain the performance benefits of using raw sockets while providing a safe interface for interacting with them. While the use of unsafe code introduces some risks, it is necessary to achieve the required functionality for ZMap. If we were simply translating the C code to Rust line-by-line, we would have to use unsafe code in many more places, but by encapsulating the unsafe code in a simple, safe interface, we can minimize the potential for errors and bugs elsewhere in the code. We were able to leverage our previous experience with writing networking code in C to implement this functionality in Rust, but it required more effort and care when working with the unsafe subset of the language.

```rust
pub struct RawEthSocket {
    inner:  Socket,
}

impl RawEthSocket {
  pub fn sendto(
    &self,
    buf: &[u8],
    interface_index: i32,
    address: &MacAddress,
  ) -> Result<(), std::io::Error> {
    // Construct sockaddr_ll struct
    let result = unsafe {
      sendto(
        self.inner.as_raw_fd(),
        buf.as_ptr() as *const c_void,
        buf.len(),
        0,
        &sockaddr as *const sockaddr_ll as *const sockaddr,
        std::mem::size_of::<sockaddr_ll>() as u32,
      )
    };
    // Check result and return
  }
}

// Usage
let packet = probe_module.make_packet(...);
let socket = RawEthSocket::new();
let res = socket.sendto(packet, if_index, &gw_mac);
```

Listing 4: This Rust code snippet demonstrates how the `RawEthSocket` struct can be used to send raw Ethernet frames safely and efficiently.

Another important aspect of ZMap is receiving packets using `libpcap`, a popular library for capturing network traffic. This also involves low-level network programming and the use of unsafe code to interact with the C library functions provided by `libpcap`. To open a "capture session", the `pcap_open_live` function is used to create a handle for capturing packets from a network interface which we also encapsulated in a safe Rust interface with a `PacketCapture` struct. Since most `libpcap` functions require a pointer to a packet capture handle (`pcap_t *`), we had to carefully think about how to manage the lifetime of the handle to avoid memory leaks and ensure that resources are properly cleaned up when the struct falls out of scope. Additionally, we cannot use the `pcap_t` struct in Rust since it is an opaque type in the `libpcap` API. Therefore, we had to take a page out of the Rustonomicon to create our own opaque type to represent the capture handle (Listing 5). This allows us to easily interact with the `libpcap` functions while ensuring type safety. We believe that working this out and creating a safe interface for interacting with `libpcap` in Rust took more effort than it would have in C. However, this is a trade-off we had to make to ensure that the encapsulated functionality is safe and easy to use in the rest of the codebase.

3.4 Blocklist

Implementing ZMap's blocklist in Rust presents unique challenges, primarily due to the language's strict ownership and borrowing

```rust
#[repr(C)]
struct pcap_t {
  _data: [u8; 0],
  _marker: core::marker::PhantomData<(
    *mut u8,
    core::marker::PhantomPinned,
  )>,
}
```

Listing 5: This struct is used to represent the `pcap_t` handle from the `libpcap` API. It is an opaque type that is used to interact with the foreign `libpcap` functions.

rules, which are designed to ensure memory safety and prevent data races. The blocklist is a prefix trie structure that needs pointers for efficient traversal and manipulation, but Rust's ownership model makes handling mutable and shared references difficult.

Rust enforces a strict borrowing system where an object can either have one mutable reference or any number of immutable references, but not both simultaneously. This system is essential for preventing data races and ensuring thread safety, but it complicates the implementation of tree structures where nodes need to reference each other, and temporary mutable references are required to modify the tree.

We implemented this prefix trie structure to handle ZMap's blocklist, which is crucial for managing IP addresses that do not want to be scanned (Figure 4). We created a `Constraint` struct that manages the whole blocklist tree, providing functionalities such as IP address lookup, counting blocked and unblocked IP addresses, and optimizing the tree for faster queries. The `TreeNode` struct represents a node in this tree and includes methods for creating a new node, checking if a node is a leaf, and converting a node to a leaf by removing its children. It contains a value (`val`) and optional references to its left and right children (Listing 6). These child

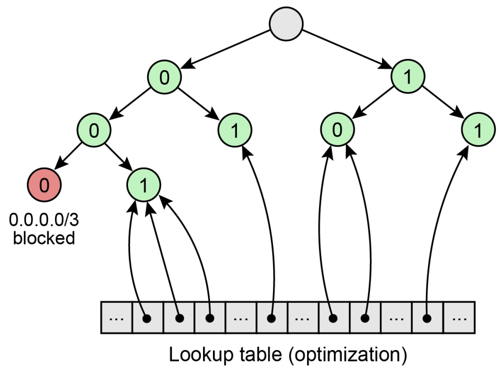

Figure 4: ZMap blocklist. This data structure is a prefix trie that is used to store IP addresses that should not be scanned. A pre-computed table of the trie at a certain prefix length is used for efficient lookups. This optimization is enabled by default. Rust's ownership model makes it difficult to implement this data structure in a simple, straightforward manner compared to a language like C. The original ZMap implementation uses raw pointers with manual memory management.

```rust
pub struct TreeNode {
  val: i32,
  left: Option<TreeNodeRef>,
  right: Option<TreeNodeRef>,
}

type TreeNodeRef = Rc<RefCell<TreeNode>>;
```

Listing 6: `TreeNode` struct definition. This struct forms the basis of the blocklist tree, representing individual nodes within the structure. References to other TreeNodes are managed using Rust's `Option` and `Rc<RefCell<T>>` built-ins, allowing for flexible handling of child nodes and ensuring memory safety and efficient traversal of the tree.

references are managed using `Rc<RefCell<TreeNode>>`, allowing multiple ownership (`Rc`) and interior mutability (`RefCell`), allowing the tree structure to be traversed and modified efficiently.

We found this data structure very challenging to implement in Rust. The original ZMap implementation was much simpler, using raw pointers for tree traversal and manipulation. However, this approach is error-prone and unsafe, as it requires manual memory management and does not provide the same level of safety guarantees as Rust's ownership model. In Rust, we had to use `Rc` and `RefCell` to manage references and ensure runtime memory safety. This approach adds some overhead due to reference counting and runtime checks, but it is necessary to ensure that the code is safe and free of memory errors. We also had to use `Option` to handle nullable references, as Rust does not allow null pointers. There is a complexity and safety trade-off when using Rust compared to C, but the safety benefits Rust provides likely outweigh the additional complexity in most cases.

4 RESULTS

To evaluate the performance of Rusty ZMap, we conducted a series of tests comparing its scanning capabilities against the original ZMap. We ran both ZMap-rs and ZMap using four threads to maintain consistency.

For our first experiment, we sent TCP SYN probes targeting port 443 to 1% of the IPv4 address space. We measured hit rate (defined as the fraction of successful responses to the number of hosts probed) with varying scan rates: 250K, 500K, 750K, 1M, 1.1M, 1.2M, 1.3M, 1.4M, and 1.5M packets per second. Each test at a specific scan rate was repeated ten times.

We also conducted a full scan on port 443 and compared it to the original ZMap under identical conditions. The primary metrics evaluated were the time taken to complete the scan and the hit rate. They both scanned at 1 Gbit/s or 1,488,095 probes per second.

Our results indicate that both ZMap-rs and the original ZMap demonstrated similar performance (Figure 5), with ZMap showing slightly more consistency in scan rates across different trials and ZMap-rs completing a full scan of the IPv4 space faster than the original ZMap. Notably, ZMap-rs had slightly higher variability in scan rates at higher packet-per-second targets. However, we maintain that ZMap-rs displays comparable performance to the original ZMap at even higher scanning rates. For the full scan of port 443, ZMap-rs completed the scan in ~42 minutes with a hit rate of 1.444%. In comparison, the original ZMap took a similar amount of time and achieved a hit rate of 1.442%.

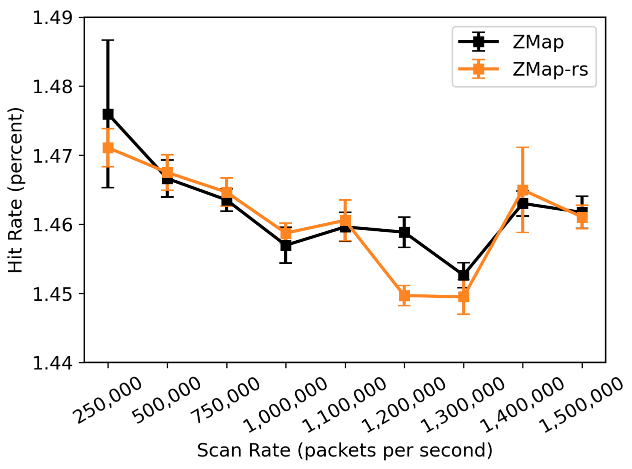

Figure 5: Hit rate vs scan rate. Mean hit rate percentages are shown with standard deviations over ten trials. ZMap-rs performs well in relation to the original ZMap. Fluctuations were observed during experimentation presumably due to the time of day. Durumeric et al. also noticed a similar diurnal effect.

From our results, we conclude that ZMap-rs is a viable alternative to the original ZMap. Even with additional overheads of concurrent state management and development complexity dealing with Rust's borrow checker and strict ownership model, ZMap-rs can achieve similar performance to the original ZMap. Our results highlight the potential of Rust for high-performance network scanning tools.

5 DISCUSSION

Our work on Rusty ZMap demonstrates that we were able to develop a tool that matches the performance of the original ZMap while providing enhanced safety and maintainability. Our experience with Rusty ZMap highlights the benefits and challenges of using Rust for systems programming tasks. Here we discuss how Rust's safety features influenced our development process and the difficulties we encountered.

5.1 Blocklist Blues

Implementing ZMap's blocklist functionality in Rust posed several significant challenges. The process required multiple iterations and an in-depth understanding of Rust's ownership, borrowing, and memory management paradigms. Here, we discuss the difficulties encountered during the development and how they were eventually resolved.

`Box`. The initial approach to implementing the blocklist aimed to mimic other tree-based data structures that people use in Rust. This version used `Box` to allocate tree nodes on the heap. However, this method encountered optimization issues and problems with reference handling. Cloning boxes would duplicate the inner tree nodes leading to inefficiencies and unnecessary memory overhead.

Stack allocation. The second version attempted to move away from heap allocation and instead focused on stack allocation. This approach aimed to simplify reference management by leveraging Rust's lifetimes more effectively. Allocating the blocklist on the stack led to problems where references did not outlive their expected scope. The borrow checker flagged errors when trying to maintain references that needed to persist beyond the local frame. Rust's lifetime annotations are important for ensuring that references are valid. However, correctly annotating lifetimes in complex data structures can be challenging, particularly when the structures need to live and be referenced in multiple frames and scopes.

`Rc` and `RefCell`. This final version successfully addressed the issues encountered in the previous iterations. The blocklist was implemented using `Rc` and `RefCell` to allow for shared ownership and interior mutability. `Rc` is a reference-counted smart pointer that allows multiple owners of the same data. `RefCell` provides interior mutability, allowing for mutable access to data even when there are immutable references to it. This combination allowed the blocklist to be traversed and mutated without running into issues with the borrow checker. This also achieves the desired performance characteristics, as individual nodes are not unnecessarily cloned. In hindsight, this was the most appropriate choice for implementing the blocklist in Rust, but it required a deep understanding of Rust's memory management and ownership model, something that may have been underestimated at the outset. This seemed unnecessarily difficult compared to ZMap's C implementation, which did not have to deal with these strict memory management rules.

5.2 Fearless Concurrency

Concurrency in Rust is a powerful feature that allows developers to write efficient and safe concurrent code. In the case of Rusty ZMap, we had to manage shared state between multiple threads to relay information about the current state of the scan to the monitor thread and for the sender or receiver threads to know when to exit. We used Rust's `Arc<Mutex<T>>` to protect shared state and ensure thread safety. This approach allowed us to explicitly prevent a data race we observed in the original ZMap implementation. However, managing shared state in Rust required careful consideration of ownership rules, as well as a good understanding of Rust's concurrency primitives.

5.3 Unsafe Encapsulation

Writing networking code in Rust was a bit more challenging than in C. In Rusty ZMap, we used raw sockets to send Ethernet frames directly to the network interface and `libpcap` to capture network traffic. This required the use of unsafe code to interact with low-level network programming APIs. While unsafe code introduces potential risks, it was necessary to match the functionality and performance of the original ZMap. We found that encapsulating well-defined unsafe code in safe interfaces allowed us to minimize potential errors and issues in the rest of our implementation. This approach also made it easier to reason about the code and ensure that unsafe code was used in a controlled and safe fashion.

5.4 Simulation for Performance Testing

During the initial stages of the project, we considered using a simulation to measure the potential speed of packet transmission. The idea was to create a virtual environment that would allow us to estimate the performance of our shared-state concurrency implementation, packet serialization, and blocklist implementation without the need for actual wide-area network activity. This approach had several theoretical advantages such as avoiding real-world packet transmission and eliminating the risk of disrupting network services. This would also allow us to easily tweak parameters and test various scenarios.

However, despite these advantages, we ultimately decided that a simulation would not meet the rigorous requirements of our project. The primary goal was to verify the practical performance of our Rust implementation in a real-world scenario, which necessitated actual packet transmission.

To meet the project requirements, we needed to conduct real-world tests by sending packets. This transition posed several challenges such as obtaining permission to use ZMap servers for our tests and engaging with ZMap network admin to ensure compliance with security policies and avoid potential disruptions. Although this demonstrated that our blocklist implementation would not adversely affect Stanford's servers or another network, it required a non-trivial amount of testing and validation to prove that our blocklist would effectively prevent scanning activities on sensitive or restricted IP addresses.

6 FUTURE WORK

Rusty ZMap has shown great promise in leveraging Rust's safety and performance features for high-speed network scanning. To further enhance its capabilities and utility, future work should consider incorporating advancements from related tools and technologies.

The latest versions of ZMap have demonstrated the capability to scan the entire IPv4 address space in as little as 5 minutes using a 10 Gbit/s connection and PF_RING. Future development of Rusty ZMap should aim to match or exceed these performance levels. This will involve optimizing network I/O operations, reducing synchronization overhead, and leveraging Rust's asynchronous capabilities to maximize throughput. We expect achieving these performance goals may present significant challenges for development in Rust, particularly in using zero-copy kernel bypass as they do in Zippier ZMap's [1]. Addressing these challenges might require the use and encapsulation of more unsafe code, as current Rust crates may not fully support these low-level features.

7 ETHICS

Similar to the ethical scanning practices in the original ZMap paper, several key steps were taken to ensure that our scanning with Rusty ZMap was conducted responsibly and transparently. Firstly, we adopted a random and unique ordering approach in our scanning process. This was achieved by implementing the cyclic scanning protocol from the original ZMap, ensuring that each scan that was run produced a different and randomized order of IP addresses, thereby minimizing the likelihood of repeatedly targeting the same IP addresses and reducing potential disruptions.

We also implemented and verified the blocklist structure from the original ZMap to prevent scanning of IP addresses that had previously opted out or were reserved. This involved testing our implementation to ensure that no blacklisted IP addresses were included in our scans. To further validate our ethical scanning practices, we conducted dry runs, simulating the scans without actually sending any packets. This allowed us to verify the randomness of the orderings and ensure that no blacklisted IP addresses were included in the results.

For our initial real-world testing, we conducted a small-scale scan of 0.1% of the IPv4 space, limiting the bandwidth to 10 Mbit/s. During these scans, we monitored the network traffic using `tcpdump` with specific filters to ensure that no packets were sent to blacklisted subnets. This was verified by running `tcpdump` in another session and ensuring that there were no hits on the blocklist filter, confirming that our scanning was compliant with ethical practices.

After verifying the dry run results and ensuring no blacklisted IP addresses were scanned, we proceeded with a real scan under controlled conditions. The `tcpdump` monitoring showed no hits on the blacklist during the real scan, which further confirmed our adherence to ethical scanning practices. With these precautions in place, we demonstrated our commitment to conducting Internet scans ethically, respecting privacy, and minimizing potential disruptions to the network and its users.

8 CONCLUSION

In this paper, we presented Rusty ZMap, a re-implementation of the high-speed Internet scanner ZMap, originally written in C, now developed in Rust. Our primary motivation was to leverage Rust's safety and concurrency features to enhance the maintainability and reliability of the original ZMap while retaining its exceptional performance characteristics.

We began by outlining the challenges associated with the original ZMap's C-based implementation, such as safety issues and complexity in managing concurrency. We then detailed our approach to re-implementing ZMap in Rust, highlighting the use of Rust's strict memory safety guarantees, type system, and concurrency features to address these challenges. Notably, we utilized Rust's `etherparse` library for packet serialization and implemented thread-safe data structures to manage concurrent state.

Overall, both implementations of ZMap achieve the goal of high-throughput internet-scale network scanning. However, the original ZMap's code base is very esoteric due to its reliance on features specific to the C language. This discourages new contributors, hurts maintainability, and creates a fragile code base that resists change. On the other hand, ZMap-rs offers a much better approach for maintainability since the Rust code is often more explicit and easier to understand. The concurrency aspect of ZMap was much easier to deal with in ZMap-rs, and should not be a deterrent for future development, since it does not lead to the code base feeling as fragile as a C codebase may. While ZMap-rs did face development challenges in areas like serialization, ownership, global state management, and low-level networking, these were all overcome with relative ease and determination. We believe this highlights the attractiveness of Rust for high-performance and memory-safe systems.

Our performance evaluation demonstrated that ZMap-rs matches and, in some cases, slightly exceeds the performance of the original ZMap. Specifically, ZMap-rs completed a full scan of port 443 in under 45 minutes with a hit rate of 1.444%, compared to the original ZMap's 1.442%. These results indicate that ZMap-rs maintains the high-speed scanning capabilities of its predecessor and also offers the same efficiency in packet transmission.

Rusty ZMap highlights the potential of Rust as a language for developing high-performance, reliable, and maintainable systems software. Our work underscores the feasibility and benefits of rewriting existing high-performance tools in Rust, providing a safer and more maintainable alternative without compromising on performance.

REFERENCES

[1] David Adrian, Zakir Durumeric, Gulshan Singh, and J. Alex Halderman. 2014. Zippier ZMap: Internet-Wide Scanning at 10 Gbps. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)*. USENIX Association, San Diego, CA. https://www.usenix.org/conference/woot14/workshop-program/presentation/adrian

[2] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. 2013. ZMap: Fast Internet-wide Scanning and Its Security Applications. In *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX Association, Washington, D.C., 605–620. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/durumeric

[3] Liz Izhikevich, Renata Teixeira, and Zakir Durumeric. 2021. LZR: Identifying Unexpected Internet Services. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 3111–3128. https://www.usenix.org/conference/usenixsecurity21/presentation/izhikevich