UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science

ReScan: High performance mass scan responder

Niels Overkamp

M.Sc. Thesis
April 2024

Examiners:
dr. R. Holz
dr.ing. F.W. Hahn

Design and Analysis of Communication Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Summary

It is currently challenging to let students run internet-wide scans using real tools such as ZMap or MASSCAN. If they were to perform these scans on the real internet they will not all obtain the same results, making assessment hard and not scalable for larger groups of students. Furthermore, we believe it to be beneficial to provide a virtual environment for students to experiment in before interacting with real-world networks, machines, network operators and internet service providers and cause real-world consequences. No such simulated environment exists for the internet-wide scale these tools operate on. Therefore we propose an approach to create a virtual internet that can intercept and respond to probes at the scale and the speed of these mass scan tools. We create a working version of this approach and experimentally verify that this approach sufficiently reaches its goals.

Contents

List of acronyms

NIC	Network Interface Card
LPM	Longest Prefix Matching
Mpps	Mega (million) packets per second
ACK	TCP Acknowledgement packet
BPF	Berkley Packet Filter
eBPF	extended Berkley Packet Filter
cBPF	classic Berkley Packet Filter
NFV	Network Function Virtualization
XDP	eXpress Data Path
AF_XDP	Address Family: XDP
DPDK	Data Plane Development Kit

Chapter 1

Introduction

ZMap [1], MASSCAN [2], scanrand [3] and unicornscan [4] are a few examples of tools used to scan large amounts of endpoints on the internet. In this research, we will call them mass scan tools. These mass scan tools are used for research: to find vulnerabilities, measure protocol adoption, measure application usage and more. However, due to their high performance, they can also be used for more dynamic research such as analyzing how reachability changes over a day.

To prepare students for such research and investigation of the internet and to help increase understanding of the internet itself it is desirable to teach the use of these tools in an educational environment. This however provides us with two problems. Firstly mass scan tools are powerful tools and a misconfiguration can flood a host, subnet or link with a significant amount of traffic, but checking students' configuration is not a scalable solution. Secondly, checking for the correctness of the results from a large group of students is not feasible since performing the scans at different times or from different locations can render different results.

Therefore we propose a sandbox environment which can act as the internet from the perspective of the mass scan tools. Using this environment one can run scans without interacting with the internet at all and with a static set of responsive endpoints. This can also be a useful tool to test configurations for anyone using mass scan tools. This document outlines the design of our approach to achieve this sandbox, as well as an evaluation of its performance and functioning.

1.1 Research Questions

A program that can reply to mass scan tools needs to perform the following actions. It needs to intercept the probes sent out by the mass scan tool. Then it needs to filter out the probes that should be responded to and finally it should construct a reply packet and send it back. These components together would appear to the tool as the behaviour of the internet and thus the goal of this research is to answer the question:

How to make a virtual responder that can intercept, filter and respond to probes at the scale and speed of mass scan tools?

1.2 Outline

First, in Chapter 2 we give an explanation for the options and working of the components our approach is built on top of, which we then use in Chapter 3 to describe the requirements and goals, and how we then got to the final design of our approach. This chapter also describes the methodology of the evaluation of this final design, of which the results can be found in Chapter 4. We discuss these results and how they match the previously established requirements and goals and where improvements can be made. We finally conclude this research in Chapter 6.

1.3 Availability of code

We have made the code available to inspect and use on the git server of the University of Twente.[1] This repository contains the Rust based XDP code, as well as the Rust code used to create filters, compile the XDP programs and load them into the kernel. As well as the code used to automate the evaluation. In the `README` the requirements are listed and the important commands are explained. The code is provided with a BSD 3-Clause License.

[1] `https://gitlab.utwente.nl/dacs/rescan`

Chapter 2

Background

2.1 Mass scan tools

We define a mass scan tool as a tool that performs scans on networks, typically the internet. These scans send probes to a range of IP addresses and possibly UDP or TCP ports. These probes are network packets that could incur a response from a host, such as a TCP SYN packet, or an ICMP Echo request or a different protocol over UDP with a response mechanism. A critical part of these tools is that they perform these scans at very high rates, reaching in the order of a million packets per second on commodity hardware. This allows these scans to cover the entire IPv4 address space within an hour on a consumer desktop.

The by us known examples of these tools are **ZMap** [1], [5], **MASSCAN** [2], **scanrand** [3] and **unicornscan** [4]. Providing a comprehensive overview of the workings of multiple mass scan tools is outside of the scope of this paper, but the concept of these tools is the same. We will focus on the working of ZMap as a case study to understand how these tools achieve their high rate and what other properties and considerations they have.

ZMap does not keep track of which probes it has sent, but only from which hosts it has gotten a reply. This makes it faster than the more conventional NMap which keeps a connection state for each host it contacts and thus requires more resources per destination and this makes it slower. A different mechanic is that ZMap attempts to not oversaturate a network by sending the probes in a randomized order. This decreases the likelihood of probes being dropped by an oversaturated link or affecting a network's reachability by overwhelming a piece of hardware.

2.2 I/O Frameworks

In the requirements section 3.1.1 later in this document we will establish the need to receive or intercept packets and to send a reply back. For this we need the ability to receive and construct custom IP packets to act as a remote host without the OS interfering. However, to obtain high performance we need to do these actions with as little overhead as possible. There are several existing frameworks, tools and approaches that give low-level networking access. In this review, we have studied the libpcap, DPDK, netmap and XDP frameworks. These are the only works we found that are receiving active support. We will summarize how these work and how they compare to each other on performance and portability, but also extensibility, maintainability and security.

According to their GitHub page [6] **libpcap** is ”a system-independent interface for user-level packet capture”. It is used by default by the mass scan tools ZMap and MASSCAN for their probe I/O. It has a high portability because of the system-independent interface allowing it to run with the same code on Windows, Linux and other systems. For the mass scan tools, libpcap is sufficiently performant for receiving probe responses on commodity hardware [5]. However, it is important to note that this does not symmetrically imply that it is sufficiently performant for intercepting all outgoing probes. The scanners receive responses for only a fraction of the traffic they generate, while this would need to process every probe. Therefore we expect that it will not perform well enough for our purposes, and since the goal of the research is not to create a comprehensive test of all possible approaches we decided not to use libpcap in the interest of time.

High-performance custom network I/O is a problem that can be found in Network Function Virtualization (NFV). This is the field of research attempting to replace dedicated networking equipment with commodity hardware running dedicated software. Zhang et al. studied state-of-the-art software switches that are used for NFV. [7] All of these switches achieve their high performance by bypassing the kernel networking stack using one of two frameworks called DPDK and netmap.

Data Plane Development Kit (DPDK) [8] is a highly performant framework, as shown by the high performance of the switches running on top of it as found by Zhang et al. [7] It directly interfaces with the network cards and performs all packet processing in user space. This bypasses the kernel and thereby eliminates all kernel overhead and packet copying. However, it does also eliminate the advantages of a kernel, namely security and compatibility. A Network Interface Card (NIC) can access all memory locations and thus a misbehaving DPDK user application can have security implications. Furthermore, custom drivers are required for every NIC and thus DPDK might not be compatible with all systems that a typical kernel would be and therefore we chose to not use DPDK.

netmap [9] solves some of the problems DPDK has by employing a hybrid approach of bypassing the kernel for per-packet processing but using the kernel to perform security-sensitive tasks such as allocating the packet buffers. netmap is not as performant as DPDK because of this interplay of kernel and userspace but does provide transmission rates of over 10 Mega (million) packets per second (Mpps). Furthermore the kernel module is not in Linux kernels and requires a recompilation of the kernel to be used.

The last approach that we are aware of is called **eXpress Data Path (XDP)** [10]. This is a feature of Linux which allows custom packet processing code to be run inside of the kernel. It does not quite achieve the same performance as DPDK. In their single-core tests DPDK reaches ∼45Mpps and XDP ∼25Mpps, but it does significantly outperform the default kernel networking stack which reaches ∼5Mpps. This means that we can achieve fast processing while retaining all of the kernel security and compatibility advantages. The downside is that the kernel needs to be able to verify that the packet processing program is safe and unable to interfere with other kernel memory areas. This means that an XDP program is limited in what it can do compared to a user space program working with netmap or DPDK.

A potential solution to this is another Linux feature called **Address Family: XDP (AF_XDP)** [11]. This adds the possibility to directly receive packets in user space from XDP, with the possibility of doing so with zero copying overhead. The receiving performance reached with this on server hardware ranges from 15 Mpps to 40 Mpps, depending on which optimizations are used. This is on par with the performance of DPDK and sufficient for our use case. Furthermore, similar to XDP itself, it is available in mainline Linux kernels since version 4.18. [12]

2.3 IP Filters

The second critical component of ReScan is determining which probes the tool should respond to. Doing this filtering efficiently is not a trivial task since mass scan tools can run on very large IP spaces, such as the entire IPv4 space. However, it seems achievable to do this in relatively low space and time since it is expected that the data has structure. For instance, certain subnets will be entirely reachable or unreachable on a certain port due to a firewall around that subnet, or due to multiple IPs pointing to the same machine. This makes the data compressible which could aid in lower space requirements and faster lookup times.

We have studied multiple such options, but decided to only employ the Bloom filter and the bitmap in our final approach. A more detailed explanation for this is given in subsection 3.1.3. We decided to include the literature research into options for filter data structures to be used for future work. To this end we will first discuss two unstructured options and how they could be applicable, followed by an overview of some structured options.

2.3.1 Unstructured Filters

The fastest option in complexity is using a bitmap or array, where a key is directly mapped to a memory address storing a boolean value. However, this causes the memory required to be $O(2^n)$ with the n being the width of the key, making it not practical for scans of bigger subnets.

A hashset computes the hash of the key and maps that to a memory address. This address contains an additional data structure to resolve hash conflicts. This then contains the boolean values indicating membership of the set. A hashset also does not take into account the structure of the scan data but allows for a tradeoff between time and space complexity by changing the hash length. Furthermore, the space required to achieve a good time complexity in a hashset scales with the number of elements in the set. This already makes it more space efficient than a bitmap since the data is sparse. Experimentation will have to show whether a hashset can achieve the speeds desired.

2.3.2 Structured Filters using LPM

For a structured solution we first establish that our IP filter problem shares properties with IP Longest Prefix Matching (LPM) routing. Routing requires a fast algorithm that can determine which port to route packets to based on its IP-address and a routing table. These algorithms can be directly applied to filter the probe packets from the mass scan tools by 'routing' them to be dropped or passed on based on the dataset. We also expect the mass scan data to structure itself based on subnets and thus longest prefix rules are an effective structure for this data.

First, we give a brief background on longest prefix matching. Then we have a look at the options from a work by Ruiz-Sánchez et al. [13] surveying various existing lookup algorithms. Finally we take a look at the shape-shifting trie by Song et al. [14] and at using Bloom filters by Dharmapurikar et al. [15]. We will not discuss how the updating works with these algorithms since our filters are static.

Longest Prefix Matching

An IP address can be represented as a series of bits, 32 bits for IPv4 and 128 for IPv6. A subnet is a collection of IP addresses sharing some amount of initial bits, called the prefix. An IPv4 subnet can be written using 4 octets and a 'mask', for

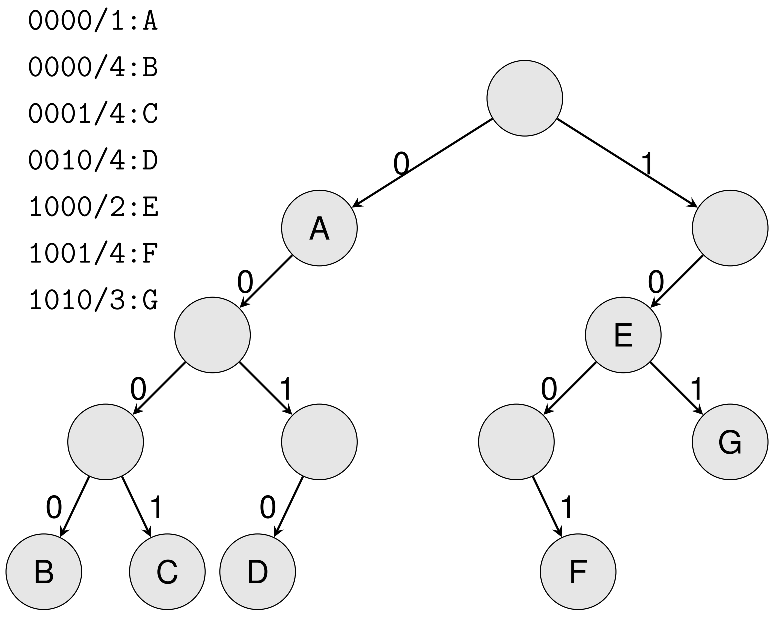


Figure 2.1: An example of a trie encoding forwarding information for prefixes of addresses of length 4

instance, `192.168.0.0/16` or `10.128.0.0/9`. The mask, the number after the slash, indicates the length of the prefix that the IPs in the subnet share.

IP routing is based on subnets since typically IPs that share a prefix are located near each other in the network graph. A router might have the routing rule that `192.168.0.0/16` should be sent from port 1 and that `192.168.5.0/24` should be routed to port 2. However, now two rules match the IP `192.168.5.2`. To resolve this ambiguity routers follow the rules of LPM and will choose the matching rule with the longest prefix, routing this IP to port 2.

Tries

The most straightforward way to perform LPM is using a binary trie. A trie is a tree where the branches represent a choice of bits in, for instance, an IP address. You can see an example in figure 2.1, for a few binary addresses of length 4. To find the longest matching prefix for a given address, the tree is traversed using the bits of the address and the value at the matching leaf is taken as the resulting value. If however, while traversing, a branch matching the current bit does not exist the algorithm terminates early and returns the last value it encountered in a node.

For example, the address `0011/4` does not have a matching leaf. While traversing the trie the last value it encounters is `A` and thus that is the resulting value. The address `1011/4` encounters both `E` and `G`.`G` is encountered last and therefore has the longest matching prefix.

In routing, tries are typically used to determine over which port a packet heading to a certain IP should be sent. However, if we use the trie to store boolean values, we can use it to define a set. If in our example the value of `A` is `True` and the values of `B-D` are `False`, then this efficiently defines the set of all IPs in the subnet `0000/1` except for the IPs `0000`, `0001` and `0010`.

Optimizing tries

Ruiz-Sánchez et al. [13] present several methods to optimize the basic binary trie. Path compression collapses all nodes with a single child. Since they have only one child, their parents node could have pointed towards their child node. This does mean that nodes need to store which bit they are deciding on since this is no longer equal to the depth of the node.

The multibit trie has each branch decide on multiple bits at the same time. This reduces the depth of the trie and therefore the number of memory lookups that need to happen. The number of bits that are decided on each branch is called the stride of the multibit trie. The prefixes may need to be expanded to fit in a multibit trie. For instance, a stride of 4 means that each prefix needs to be a multiple of 4 and therefore a prefix of length 9 needs to be expanded to 8 prefixes of length 12. Choosing a stride creates a tradeoff between lookup time and memory usage.

This paper presents a few more solutions using tries, combining path compression and multibit techniques. However, these do not provide an improvement in the worst-case lookup or memory usage while adding significant implementation complexity and therefore we will focus on the binary search techniques.

Faster lookup in address width

The basic and optimized versions of tries all have a lookup complexity linear to the width of the address. The tries using multibit strides reduce the complexity by a factor equal to the size of the stride, but this is still linear with the width of the address. Furthermore, the memory complexity increases exponentially with the size of the stride. The main consequence of this is that an algorithm that performs well for IPv4 might not perform well for IPv6 since it quadruples the address width. Even though IPv6 support is not a main goal we do include some algorithms that scale better to keep the option to add IPv6 open. The first two are from the survey paper by Ruiz-Sánchez et al. [13], and the latter are two from individual studies.

The first solution considers that an LPM algorithm performs two lookup operations: one to find the matching prefixes and a second to find the longest one. The binary search on length method performs a binary search to determine the longest prefix that matches. It achieves this by splitting the different prefix lengths into hash tables and performing a binary search over these hash tables. If it finds a match in a hash table it continues the search in the longer prefixes, and otherwise in the shorter prefixes. This does require the computation of so-called markers to guide the search in the case where a longer prefix exists, but no match does. It obtains a worst-case lookup speed of $O(log_2(W))$ where W is the width of the address used. This is because of the binary search on the address width.

The second solution considers the ordered list of IP to value mappings, and that the prefixes shorter than the full length create long continuous ranges of the same value. The solution is called binary range search and it takes these ranges of continuous values and creates a binary tree to match an IP to the range it falls in. Note that a prefix does not correspond one-to-one to a range. For instance, two 4 bit IP prefixes of `1000/2` and `1001/4` correspond to three ranges: `[1000, 1000]:1000/2`, `[1001,1001]:1001/4` and `[1010, 1011]:1000/2`. This solution does not depend directly on the address width and as such has a worst-case lookup speed of $O(log_2(N))$ where N is the number of prefixes.

The third solution [14] uses the sparse nature of IPv6 tries to encode the data more efficiently and has the tries 'shape-shift' to reduce the number of nodes that need to be traversed. The last [15] uses multiple parallel Bloom filter lookups for different address lengths to find the LPM, and then a hashmap lookup for the longest prefix match found. Best case this would mean 2 lookups, but Bloom filters can give false positives, and as such the amount of lookups depends problematically on the size of the Bloom filters.

Chapter 3

Design & Evaluation Methodology

In this chapter we describe how we arrived at our final approach to respond locally to the probes of mass scan tools. We created a prototype, then the final tool, and finally evaluated it. This chapter describes this process.

3.1 Design

3.1.1 Requirements

To be able to respond to scans as performed by the tools expanded upon in 2.1 ReScan needs to be able to do four operations. It needs to intercept or receive the probes sent out by the scan tool, then filter out the ones that require a reply. These filtered probes then need a reply to be constructed which finally needs to be sent back to the scan tool.

Furthermore, as the focus of this research is to present the viability of the approach taken by ReScan, and to provide a working tool, it should support at least a minimum use case. The use case we have chosen is a TCP ACK scan on IPv4, as this is a very common scan to perform. However, it should not be limited to either TCP scans or IPv4 and should be designed with extensibility in mind.

With these considerations we can construct the following list of requirements:

1. Receive or intercept packets from mass scan tools
2. Filter probes based on a given dataset of destinations
3. Construct a reply with a message appropriate to the protocol(s) used
4. Send reply back to the mass scan tool
5. Support TCP SYN scans
6. Support full IPv4 scans

7. Be extendable to other types of scans

Besides these requirements we also constructed a few goals:

1. ReScan should be performant; it should be able to run close to the speeds that the mass scan tools run on the internet
2. ReScan should be easy to run on different hardware; e.g. consumer hardware or servers

3.1.2 Prototype using libpcap

We first created a prototype to assert that running mass scans locally without probes ever leaving the machine would be feasible. This approach uses libpcap to capture the packets and to send the replies. A Rust program is used to parse, filter the probes and to construct a reply.

The Rust pcap library is able to receive all packets sent out on the loopback interface. TCP packets received this way then have their destination IP matched against a HashSet. If the IP is present in this HashSet a spoof reply is constructed and sent back out onto the loopback interface. If the reply is well-formed it is received and interpreted by the scan tool as a reply from the destination host and thus this approach would result in the scan tool reporting all the hosts in the HashSet as online and reachable.

Creating a spoof reply requires creating a reply as the destination host could create it. This means swapping the source and destination address and port, and furthermore making it a valid TCP SYN ACK response.

This approach fulfilled all of the functional requirements and also is easy to run on many different machines as libpcap is available on many operating systems and hardware. But from the literature and from empirical tests it became clear that it would not reach the performance desired.

3.1.3 Approach with XDP

To reach higher packet processing speeds we researched multiple frameworks for efficient network I/O. The more detailed findings and analysis can be found in 2.2. libpcap is expected to not be able to process the probes at the speed at which the mass scan tools send them. In the introductory paper of ZMap [5], the use of libpcap is also mentioned as a potential bottleneck and only used because it only needs to receive a fraction of the probes it sends out. DPDK has a high performance but potential security concerns, by bypassing the kernel it also bypasses the kernel security features regarding its network infrastructure. Because of these reasons we decided to make a prototype with XDP and with netmap, for the reported packet processing speeds without needing to bypass the kernel.

However, working with both it became clear that using netmap was significantly more of a challenge than XDP. Where for XDP there is adequate documentation and we got a simple program up and running within a day, netmap proved to be hard to implement and would require a large amount of expertise in Linux kernels which reduces possibility of extension by others in the future. Furthermore the requirement to modify the kernel to be able to run on Linux machines makes it much less usable. Therefore we chose to follow through with XDP.

XDP, for eXpress Data Path, is a method of running custom extended Berkley Packet Filter (eBPF) code at the head of the kernel networking stack. Incoming packets can be dropped, rerouted or passed on to the networking stack. Since XDP only captures incoming packets we are again making use of the loopback interface to route the outgoing probes of the mass scan tool into XDP. To filter the probes we initially used the hash map Berkley Packet Filter (BPF) map. BPF maps are the main way to share data from user space to the XDP program, allowing both user space and the XDP program to read and write to this hash map. We use it to map IP addresses to boolean values to indicate whether a given host is online.

To clarify, the acronym BPF, is used in this text interchangeably with eBPF. This is because eBPF still refers to its data structures as 'BPF maps' in the documentation and code. We do not use what used to be referred to with BPF and is nowadays called classic Berkley Packet Filter (cBPF), as this is no longer being used in current Linux Kernels.

Contrary to the libpcap approach we do not construct a reply from scratch at this point. Instead we modify the following header fields of the probe to morph it into a valid reply, since many fields can simply stay the same. We swap the IP addresses, TCP ports and acknowledgement and sequence number. We do then need to increment the acknowledgement number by one and set the TCP Acknowledgement packet (ACK) flag. Besides saving time constructing a full packet this approach makes the checksum computation very fast. The exact algorithms for packet altering can be found in Listing B.2 - Listing B.4.

TCP over IPv4 uses a checksum that is the *"16-bit ones' complement of the ones' complement sum of all 16-bit words in the header and text"* (from RFC 9293) [16]. Since these 16-bit words are aligned with all the values we swapped, the swapping does not change the value of the checksum. The incremented acknowledgement number and the added ACK flag do, but we can directly compute the new checksum. We can invert the old checksum to obtain the sum of the 16-bit words of the old header and text, add 1 for the incremented acknowledgement number and add the value of the ACK flag (16). If we then invert this value we get the new checksum.

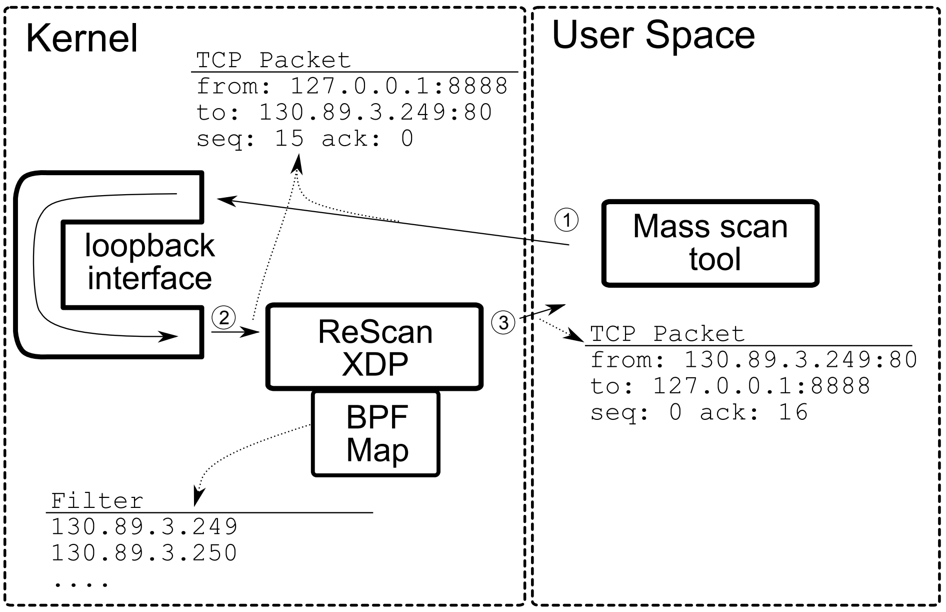


Figure 3.1: Diagram showing the flow of an example probe through ReScan. Only part of the TCP header is shown as an example. 1) The packet is sent by the mass scan tool. 2) The packet triggers an XDP execution, filtering and creating the reply. 3) The reply is passed through the network stack back to the mass scan tool

With add we mean a 16-bit ones' complement addition. Further, in case of an overflow of the least significant 16 bits of the acknowledgement number the ones' complement sum of the acknowledgement number does not change. In this case we only add the value of the ACK flag.

After having created the reply from the probe we can simply pass it on to the networking stack which will deliver it to the mass scan tool. We paired this XDP program with a user space program that loads the XDP into the kernel and then fills the BPF map with IPs from a dataset given by the user. This makes this approach fulfill requirements 1-5, and since we have split out the the handling of the Ethernet and IP layer, it is relatively easy to create a BPF program for a different kind of scan, fulfilling requirement 7.

However, we found through initial testing that when we would try to create a filter from an IP dataset from a typical full IPv4 TCP scan with the tens of millions IPs, it takes multiple minutes on commodity hardware to fill the BPF map. Although this is not a long time in comparison to the duration of a full IPv4 scan, this process needs to complete before the scan can be started. For instance when mistakenly selecting the wrong dataset or running a small scan against a large dataset this would cause unnecessary delay. Especially since we expected that filling the map with tens of millions of IPs could be much faster, since it is essentially writing just a few hundred

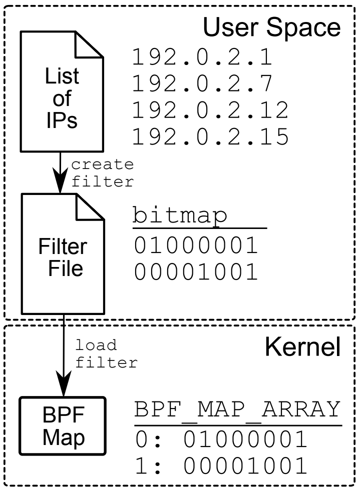


Figure 3.2: Diagram showing the creation and loading of an example dataset, in subnet `192.0.2.0/28`. In the `create filter` step the list of IPs is converted to a binary representation. The `1`s in the bitmap directly correspond to the IPs in the list, and these two bytes correspond to the full `/28` subnet. `192.0.2.1` corresponds to the `1` in the second position and `192.0.2.15` to the `1` in the last. In the `load filter` step the data from the filter file is loaded into the BPF Map. For the sake of this example it uses a chunk size of 8 bits instead of 32KB.

megabytes, which should not need to take minutes. Therefore we decided to look into speeding up this process.

Batch Insertion

The long insertion times are most likely caused by every insertion triggering a context switch from user space to kernel space. The logical solution to this would be to batch the insertions to reduce the amount of context switches. BPF maps do support batch insertions, however, it is not enabled by default and would require a recompilation of the kernel. This is undesirable since it would make this approach less portable and not fulfill our second goal.

The alternative we found was to encode multiple IP addresses in a single inserted value. Since a single element in a BPF map can be as large as 32KB, this would significantly improve the time it takes to create the filter. To further improve filter loading times we added a preprocessing step where we convert the plain text list of IPs into a binary representation of our dataset. Upon loading the filter into the BPF map we can simply read this binary data from the file, in chunks of 32KB and insert these chunks into a BPF array map.

Filter data structure

We implemented two data structures to be encoded in this way. The first is a bitmap, a bit array with each index in the array representing the IP address encoded by that index, and the bit at that index represents whether the IP is in the set or not. This takes up a considerable amount of space but lookup is $O(1)$. The second is a Bloom filter to reduce the amount of data we would need to send to the kernel and keep in memory. A Bloom filter is a statistical data structure to represent a set. It uses multiple hashes to ensure that it will not report false negatives, but there is a chance that it will report false positives.

We decided to not implement any of the structured filters that are discussed in subsection 2.3.2, but to perform the evaluation with the abovementioned unstructured data structures. This choice was made because implementing these more complex filters is not required for the requirements or the goals of ReScan, and because of time constraints.

3.2 Evaluation Methodology

To show the correct working and to evaluate the performance goal we ran a series of test on two different machines, with different parameters and datasets.

3.2.1 Measurement parameters

The main measurements we are taking are:

- the transmission rate as reported by the mass scan tool
- the amount of IP addresses received that should not have been received, which we call the false positives
- the amount of IP addresses that were not received but should have been, which we call the false negatives or drops

The transmission rate informs us about the performance of our approach when run together with the mass scan tool, while the false positives and drops inform us about the correctness.

Additionally to these measurements we are also measuring the runtime of the XDP program. Lastly we collect the filter creation time to evaluate whether this is within an acceptable range.

Server		AMD EPYC 7F72 24-Core Processor 256GB RAM (4x64GB) DDR4 3200MT/s *Kernel:* 5.10.0-18-amd64 *OS:* Debian 5.10.140-1
VM	**Host**	Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz 188GB RAM (12x16GB, dual channel 6x2) DDR4 2133 MHz *Kernel:* 4.15.0-58-generic *Hypervisor:* KVM *OS:* Ubuntu 18.04
	Virtual	8 Virtual CPU's 16GB RAM *Kernel:* 5.10.0-9-cloud-amd64 *OS:* Debian 5.10.70-1

Table 3.1: Hardware specification

3.2.2 Measurement setup

First we will describe the parameters we chose or varied and explain our choices.

Machines We are using two machines as hosts to perform the tests, each running both ZMap and the XDP program, as is the intended use. The first machine was chosen to represent a typical use case of running the program on a VM, for instance assigned to students. This is a fairly standard machine with 8 virtual CPU's and 16GB of RAM, and we will call it the 'VM' from here on. The second machine was chosen to test the limits of the program with a root server machine with 24 cores, 48 threads and 256GB of RAM. This one will be called 'server'. The relevant information on the machine hardware can be found in table 3.1.

Threads To make use of hyperthreading we need to assign a number of threads to be sender threads for ZMap. Since ZMap uses 1 thread for receiving and 1 thread for monitoring we chose to allow ZMap to use all remaining threads for sending. For the VM this is 6 and for the server 46. This might seem unbalanced, however as we explain in the 'datasets' paragraph, only about 1 in 70 sent probes receive a reply. Therefore a 46 to 1 ratio of sending and receiving threads is reasonable.

A second seeming issue with this division is that it leaves no threads for running the XDP code. However, experimentation showed that there was no discernible performance increase when using fewer send threads for ZMap.

Datasets We selected two datasets, both from a ZMap TCP SYN scan on a single port. The first was performed specifically for this research and is a full IPv4 scan on port 80, this has a hitrate of 1.43%. We will call this the 'dense dataset' or the 'port 80 dataset' in this research. The second is from a different ongoing research project, which did a full IPv4 scan on port 389, which is the default port on which

LDAP is hosted. This has a hitrate of 0.11% and will be refered to as the 'sparse dataset' or the 'port 389 dataset'.

A higher hitrate in the dataset will correspond to a higher number of probes passing through the filter, and thus would need more time to be processed by XDP. It also means a higher number of probes that need to be processed by the kernel network stack and then by the mass scan tool. Using two different datasets might give us insight in how this might affect our performance goal.

Subnet size Ideally we would run all tests on the full IPv4 range, but this causes the tests to be significantly longer. We decided to do all tests on the server on the full `/0` range, and the tests on the VM on a random `/8` range to keep the runtime of the testsuite within doable ranges. This was because we found that the test on the VM on a `/8` subnet for a single dataset took about 19 minutes, which would mean that it would take about 82 hours to complete the test on a `/0` range, and even more for both datasets. Running this test multiple times would mean that it would take weeks to complete.

This did present the additional problem of choosing a subnet from the dataset to run the VM on. We wanted to keep the hitrate of the dataset used for the VM similar enough to the one used for the server to be able to compare the results. To solve this we choose a random subnet, and then if the hitrate of chosen subnet is off by more than 25% we try again with a different random subnet.

It is also worth noting that in the full IPv4 range we exclude the `127.0.0.0/8` subnet as to not have the scan traffic mix with the normal traffic on the loopback interface.

Filter type As described in section 3.1.3 we want to evaluate both filter options we created. These are the Bloom filter and the bitmap. The size and amount of hash functions of the Bloom filter are chosen such that it would theoretically produce a false positive rate of 0.001. For this we use the following formulas, where k is the number of hash function, p the false positive rate, n the amount of IPs to be filtered and m the size of the Bloom filter in bits. Equation 3.2 describes the optimal value for the Bloom filter size given the size of the set of IPs and number of hash functions. Equation 3.1 describes the approximate relation between the number of hash functions and the false positive rate. These two formulas are from the work of Kirsch and Mitzenmacher [17].

To use these values for k and m we need to round them to be integers, and furthermore we need to round m to a power of 2. We assumed this at the construction of the Bloom filter to simply its construction and lookup. For the round to a power of 2 we are using 3.3.

$$k = -log_2\, p \tag{3.1}$$

$$m = n\, \frac{k}{ln\ 2} \tag{3.2}$$

$$round^*(x) = 2^{round(log_2(x))} \tag{3.3}$$

Mass scan tool We chose to use ZMap to perform our performance evaluation because it is the primary target for ReScan. We run ZMap with multiple transmission threads, 46 for the server and 6 for the VM. To obtain a baseline performance of ZMap on the specific machine with a specific subnet size we have it run against a dummy interface first. The dummy interface is a software network device that drops all packets given.

Transmission rate Finally, to study how a higher transmission rate might affect drop rates we run the tests with giving ZMap different transmission rate limits. ZMap will attempt to not exceed this limit and allows us to find the trade off of bandwidth and drops.

Seed For the random subnet selection as explained in the 'Subnet Size' paragraph above we use a fixed seed. This is to be able to compare the various tests. The value of this was arbitrarily chosen to be 322376503.

Now we list the different measurements we will be taking and how we will do so.

Transmission rate ZMap reports the average rate at which it conducted the scan, we read this from the standard error output of ZMap and record it.

Runtime We can set the kernel flag `bpf_stats_enabled` to collect the total runtime of the XDP program and the amount of times it has run. We can then estimate the average runtime of a single call by division. We can also use the run count as an additional measure to determine whether all probes sent by the mass scan tool is also received by XDP. Enabling BPF stats causes the runtime of the XDP program to be longer, and as such we will perform tests with it enabled and tests without to evaluate whether the other measures are affected by it.

Correctness We collect the list of IPs that ZMap reports as being online. These are the all the packets that ReScan responded to, and as such can be used to assess the correctness. We compare it to the dataset and count false positives and false negatives.

Chapter 4

Results

To assess completion of the requirements and performance of the goals as set out in 3.1.1 we set up a set of experiments as detailed in 3.2. After running these experiments we collected the data into the graphs and tables presented below. The graphs with caps display the symmetric standard deviation of the data over the multiple test that were run. This variance data can also be found in Appendix A.

4.1 Send Rate

One of the main measurements we have taken is of the transmission rate to give insight into the performance of our approach. Plotted below is the rate that ZMap reports to be be sending probes at. This is plotted against the transmission rate limit that ZMap was set at for a certain test. In figure 4.1 the test cases with both datasets is shown for the VM testbed. The test case with the dense dataset from a scan on port 80 is shown as well as the test case with the sparse dataset from a scan on port 389.

The measurements were taken from ZMap as packets per second but are displayed in the graphs as megabits per second using the fact that all probes sent have the same minimal length of a TCP packet: 58 bytes.

In all the send rate graphs the blue line shows the baseline measurement. This is the maximum ZMap can reach when all packets are dropped immediately and thus require no additional computation. The other four are specified to be either a test with bpf-stats turned on ('bpf-stats') or not ('normal'). Furthermore they are labelled to be using the bitmap filter or the Bloom filter.

All graphs show the reported send rate be equal to the set send rate up to a certain point. At this point the graphs 'flatten out' and ZMap reports approximately the same send rate for all set send rates bigger than this point.

For the VM this point lies between 550 Mb/s and 570 Mb/s for the baseline and

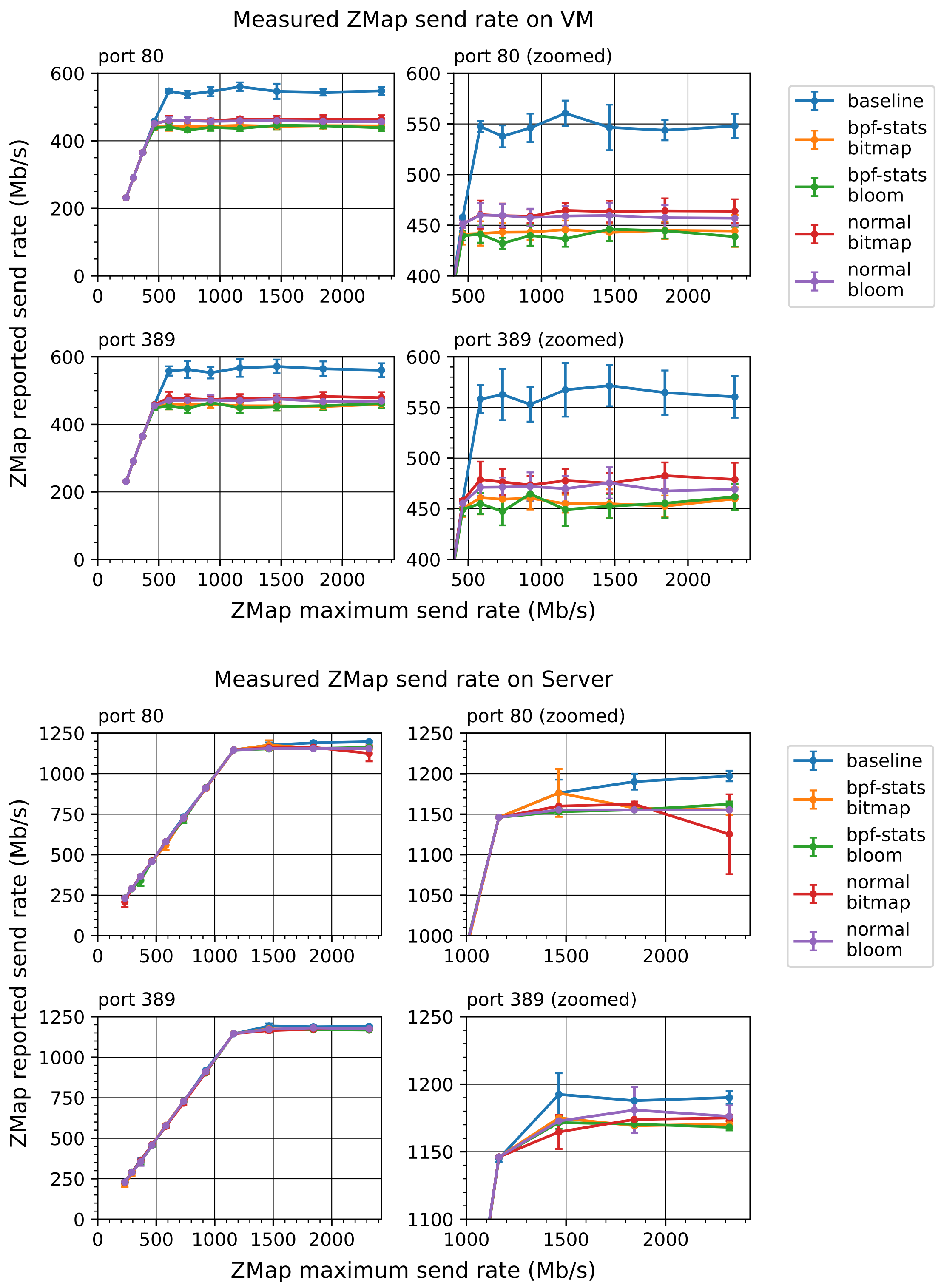


Figure 4.1: The rate at which ZMap reports to have sent out packets on the two testbeds, plotted against what rate it was set to not exceed. Plotted is the baseline test with packets being dropped immediately, a test with bpf-stats enabled and a test without bpf-stats. The latter two are run with a Bloom filter and a bitmap filter and labelled as such.

between 430 Mb/s and 480 Mb/s for the cases with XDP enabled. The tests with bpf-stats enabled perform slightly worse than the tests without bpf-stats, with the mean differing by 20 Mb/s. The bitmap filter has a send rate mean higher than the Bloom filter, but the difference is slight: about 10 Mb/s at peak.

Looking at only the test using the bitmap filter and with bpf-stats disabled, we can see that it consistently has the highest send rate for the VM, but the difference between the bitmap filter and the Bloom filter is smaller than a single standard deviation. The mean values of the flattened out part of the this test lies between 470 Mb/s and 480 Mb/s and are the maximum send rates our approach stably achieved on the VM.

For the server the flattened out part lies between 1190 Mb/s and 1200 Mb/s for the baseline and between 1150 Mb/s and 1170 Mb/s for the cases with XDP enabled. The last data point of the test with the port 80 dataset, the bitmap filter and with bpf-stats disabled, and the data point at $x = 1450Mb/s$ of the same case but with bpf-stats enabled both have a very high variation, and are likely outliers. Therefore we choose to ignore these.

The results of the server case with XDP running have closer means and higher variation than the VM case. Not one test case is clearly performing better than the others. However, ignoring the outlier, the test case that performed best on the VM, achieves a mean value at the flattened out part of 1170 Mb/s.

4.2 Processing Time

Another measurement to gain insight into the more isolated performance of our approach uses the built-in statistics module of BPF, bpf-stats. This reports the total run time of the loaded XDP program, and the amount of times it was called. We can use these to calculate the average run time of a single XDP execution. It is important to note that bpf-stats does add some overhead, which can be seen in figure 4.1. With bpf-stats enabled, the throughput is slightly lower. The only source we could find on this overhead states that it adds about 20ns of overhead per execution [18]. However, nothing further could be found to substantiate this claim.

The bitmap filter approach seems to result in a lower runtime, and also a more consistent runtime. It performs roughly the same for the sparse and dense dataset, whereas the Bloom filter performs significantly worse with the dense dataset compared to the sparse dataset, in both the server and the VM scenario.

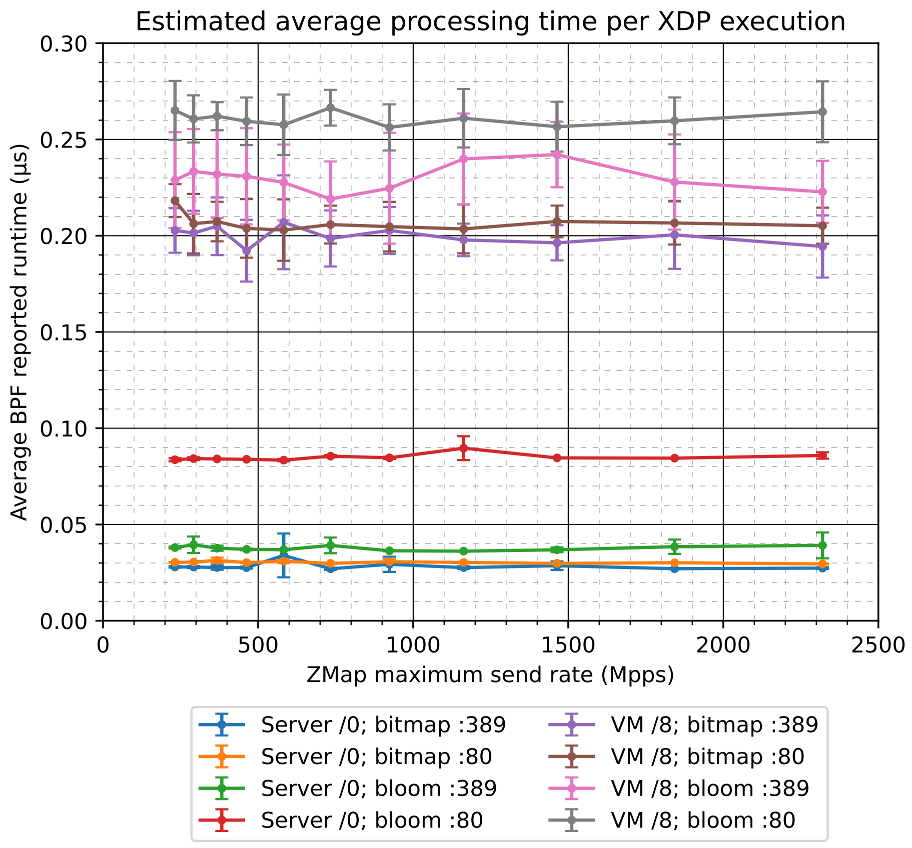


Figure 4.2: The estimated average time it took ReScan to process a single probe plotted against the send rate ZMap was set to not exceed. This uses the bpf-stats and thus only the bpf-stats tests are plotted.

4.3 Correctness

We have split up the correctness measures into a false positive and a false negative rate. These show the mismatch between the original dataset, and thus the responses that ZMap should receive, and the responses that ZMap receives. A response being received by ZMap that was not in the original dataset is called a false positive. A response not being received by ZMap that was in the original dataset is called a false negative.

Both correctness counts are shown as a ratio. The false positive rate is the ratio of the count of false positive responses divided by the amount of addresses in the subnet. A subnet of /24, consisting of 256 addresses, and a false positive *count* of 64 would result in a false positive rate of 0.25. The false negative rate is similar but divides the count of false negative responses by the amount of addresses in the *dataset*.

	Mean False Positive Rate	**SD**
VM /8 bloom; dense	0.0002098	$3.311 \cdot 10^{-6}$
VM /8 bloom; sparse	0.004638	$8.696 \cdot 10^{-5}$
Server /0 bloom; dense	$2.10369334501379 \cdot 10^{-9}$	0
Server /0 bloom; sparse	0	0
All bitmap	0	0

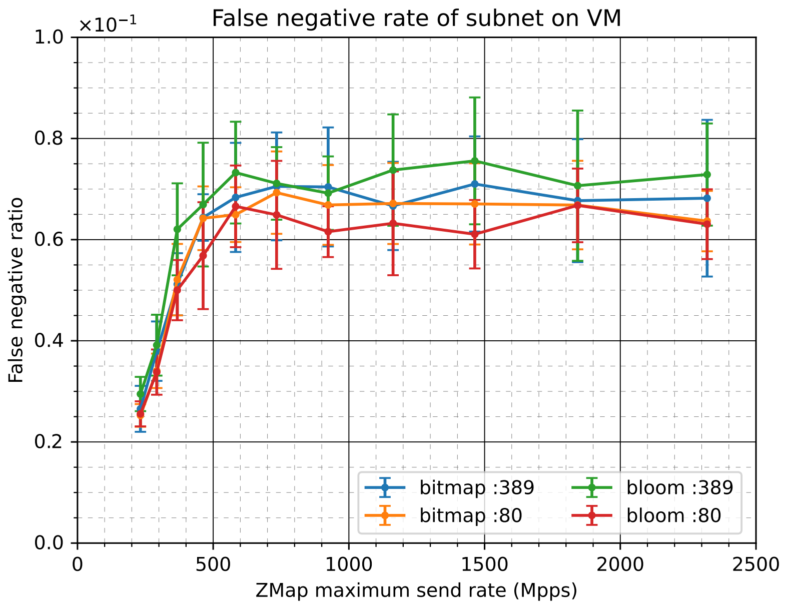
Table 4.1: The average amount of responses received that should *not* have been received, as a ratio of the total subnet size

The mean false positive rate is shown in table 4.1, with the results of multiple runs combined and the mean and standard deviation shown. False positives only occurred when using the Bloom filter, and orders of magnitude more so for the VM than for the server. The dense VM test case on port 80 reports $1.05 \cdot 10^6$ times larger false positive rate, and the sparse VM test case on port 389 an additional 22 times. Furthermore all server cases report the same number of false positives regardless of which run or the maximum transmission rate. This can be seen in the standard deviation of 0.

The mean false negative rate for the different test cases, plotted against the maximum send rate is shown in figure 4.3. For the VM test case the false negatives seem to be strongly dependent on the send rate, rising steeply as the maximum send rate rises and plateauing and the same point as the reported send rate does. At this plateaus the mean ratios lie around 0.07. This means that at higher send rates 7 out of every 100 expected probe responses are not received. The variation also rises as the false negative ratio rises, and in all cases the differences between the test cases are smaller than a single standard deviation.

The false negative rate of the server case lies close to zero, with the highest mean at $8 \cdot 10{-}6$ with the sparse bloom case and for some tests all runs having resulted in all probes arriving and thus a false negative rate of zero. For the cases with a non-zero false negative rate, the standard deviation is relatively high, often bigger than the mean. This indicates that for almost all the cases there were no false negatives, but that for a small number of cases the false negative rate lies higher. This seems to occur more often and more severely for the Bloom filter cases.

We also looked at the run count that BPF reports and we used in section 4.2. This can give us insight as to whether these errors occur more before or after the XDP code. The conclusion from the run count is that the amount of times that XDP is run is equal to the amount of probes sent out by ZMap, with only small differences. XDP was over all runs called at most 26 times more than expected, and at most 2 times less than expected. This is a negligible amount compared to the millions of probes being sent, more than 16 million for the VM and more than 4 billion for the



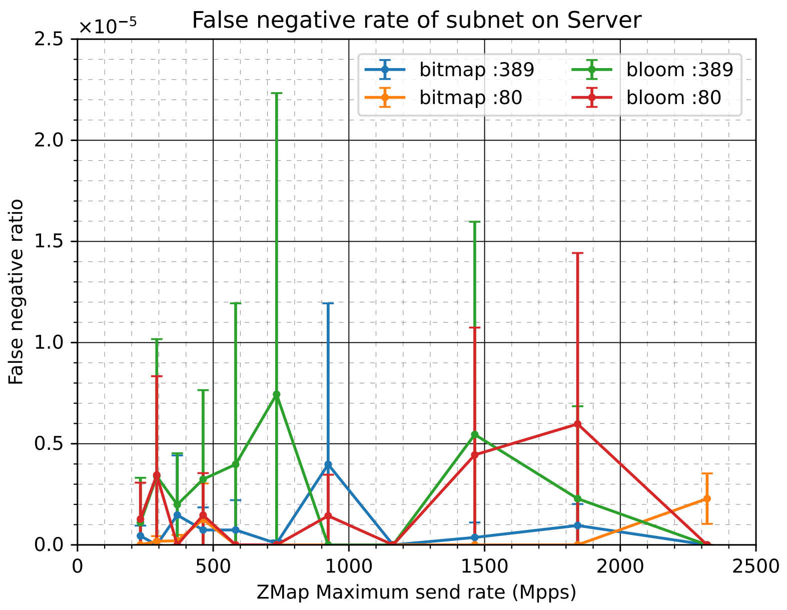


Figure 4.3: The amount of responses not received that should have been received, as a ratio of the amount that should have been received, on the VM and server testbed respectively

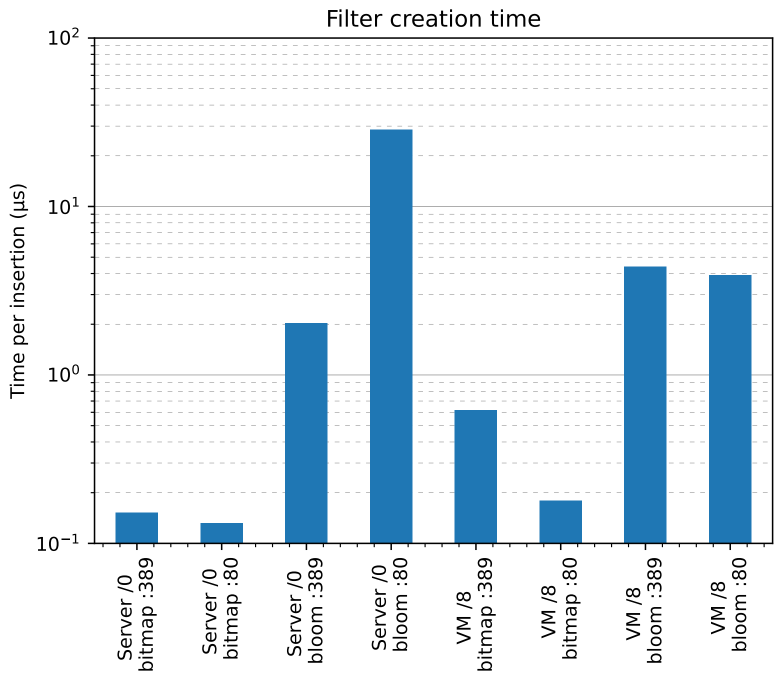


Figure 4.4: The time in microseconds it took on average to insert a single IP into the filter

server.

Because we did not expect the results from the false negative rate on the VM to be as high as it is, more on this in section 5.2.2, we performed an informal test on a personal laptop. Due to time constraints we did not fully analyze the results from this however it is relevant to mention that in these tests we saw only a handful of false negatives, similar to the behaviour of the root server.

4.4 Filter creation time

A bar graph showing the time it took to create the filter before running the test is shown in figure 4.4. The bars show the amount of microseconds it took on average to insert an IP into the filter, so the total time it took to create a filter can be calculated by multiplying this number by the number of IPs in the filter. We chose to normalize this measure like this to be able to compare the different filters with different sizes.

The worst performing filter is the Bloom filter on the server with the http scan dataset. It takes 29 microseconds to insert a single IP, corresponding to 28 hours to construct the total filter. Compared this to the 0.13 microseconds insertion time for the same case with the bitmap filter, corresponding to 8.1 seconds for the complete filter. Overall the Bloom filters perform about an order of magnitude worse than their bitmap filter counterparts. The ”VM /8 bitmap :389” case and the ”Server /0 bloom :80” seems to be outliers, both having longer insertion times than the other cases with the same filter type.

The bitmap filter’s worst performance is on the VM with the sparse dataset, with 0.65µs per insertion. If we would apply this speed to the creation of the full /0 :80 dataset with 61 million entries, it would take 36.7s in total. This is of course not entirely accurate since the size of the bitmap is dependent on the size of the subnet and this might affect the insertion speed. However, it does give a scale when compared to the worst bloom performance which took 29 hours and 6 minutes.

Chapter 5

Discussion

Here we reflect on the results as described in 4, how those results fulfill the goals and requirements as set out in 3.1.1 and we suggest the next steps for research in this tool.

5.1 Send Rate

5.1.1 Performance goal

In section 4.1 of the results we showed the difference between running ZMap with our approach enabled and if we drop all packets instead. Choosing the bitmap filter and the dense dataset allows our approach to complete a TCP scan at 460 Mb/s on our VM machine and at 1160 Mb/s on the root server. At 58 bytes per probe this results in 1.0 Mpps and 2.5 Mpps respectively. To scan the full IPv4 space it would take 1 hour and 12 minutes on the VM and 28.6 minutes on the server. If we would instead use the sparse dataset it would instead take 1 hour 9 minutes on the VM, and 28.4 minutes on the server.

Since the baseline case is only running ZMap and packets are dropped as soon as they get written to the interface we can take its final ’flattened’ value as the maximum send rate this implementation of ZMap can achieve on this machine. For the VM this maximum would be 570 Mb/s and for the server 1190 Mb/s, or 58 minutes and 27.7 minutes respectively to complete a full IPv4 scan. We assume that running ZMap on the dummy interface is faster than running it on a network card because the dummy interface drops every packet. Therefore, with at most a 11 minute penalty on a baseline of 58 minutes we can conclude that our approach is able to run close to the speeds that mass scan tools run on the internet, as our first goal stipulates.

5.1.2 ZMap Bottleneck

In this section we will show what we believe to be a likely bottleneck for ReScan to reach higher throughputs.

From the previous subsection we can see that the server completes a TCP scan about 2.4 times faster than the VM. This is lower than one might expect due to the specifications of these machines. The VM has 8 threads available, and the server 48. In theory we might expect a 6 fold speed up. In practice this increase would be lower due to multi-threading overhead such as scheduling. However, in the baseline cases the server only performs 2.2 times faster than the VM, and here we are only sending. Scheduling alone is not enough to explain this discrepancy.

In addition to this the server CPU's are also more powerful, from section 4.2 we can see that the XDP code on the server executes 3 to 7 times faster than on the VM, however this clearly does not translate linearly to the scan speed.

This both points to a bottleneck somewhere in our approach, specifically in the packet generation and sending step. This is because the lower-than-expected speed increase on the server occurs both for the baseline and the loaded tests. In the baseline we are almost exclusively generating and sending probes using ZMap, but with more than 6 times the cores, which are also more powerful, ZMap is only able to do this at 2.2 times the speed it is on the VM.

In theory the bottleneck in the baseline tests could also be occurring in the dropping of the packets on the dummy interface, or some intermediate network handling that might occur. However, in the literature [19] we can find that ZMap, when used as is, is capable of running at gigabit Ethernet speed which is limited at 1.44 Mpps. This is in ZMap's normal operation, running over a physical NIC to an Ethernet line, whereas in the baseline approach we are dropping the packets in the kernel. However, no data on the limitation of ZMap when using the loopback or dummy network interface is available. In fact, we find that in this setup ZMap is able to achieve 2.59 Mpps on the dummy interface and at least 2.50 Mpps on the loopback interface. Thus, given that ZMap as is was intended to be used at gigabit Ethernet line speed, it is highly likely that it causes a bottleneck when attempts are made to run it far above this limit. This makes ZMap being the bottleneck highly likely.

5.2 Correctness

5.2.1 False positive rate

Some false positives are expected for the cases using the Bloom filter, this is simply how the Bloom filter works. However, we designed the Bloom filters to have a false positive rate of 0.001, as described in subsection 3.2.2, choosing the Bloom filter parameters using the formulas described. None of the tests result in a false positive rate of 0.001, with the VM sparse case having a higher rate and the others a lower, with the server sparse case even having no false positive. This might be due to the hash-functions not being independent enough although we would expect this to result in only a higher false positive rate. We do not have a further explanation for this behaviour. The bitmap performs better or equally to the Bloom filter in throughput and thus it is not relevant to investigate this further.

5.2.2 False negative rate

The most surprising result from this benchmarking is the high false negative rate on the VM. We stated in section 4.3 that for every 100 expected probe responses about 7 do not arrive at ZMap when the send rate is high enough. First we suspected that this is likely due to congestion between the point where the reply is constructed and where it is received. If this was the case we would expect to see a much lower drop rate for the sparse dataset because this has a lower response rate and thus the congestion should be lower. We also stated that the send rate is strongly related to the drop rate. This made us suspect that the drops are due to congestion before the XDP code receives the probe, for instance in the outgoing traffic buffer of the loopback interface. However, we also mentioned in section 4.3 that only a negligible amount of packets are dropped before our XDP code is executed.

With data indicating that the drops are not occurring before or after our XDP code, we might conclude that it must be occurring during the XDP code execution, either due to a coding bug or because the execution is terminated before finishing. However, because this drop behaviour is not observed on the server, a coding bug seems unlikely. Furthermore terminating a running XDP process does not seem to be a logical action to take in case of congestion, we would expect skipping of some executions to not waste already performed computation.

Concluding from these seemingly contradictory indications it seems most likely to us that there is drops occurring in the incoming response traffic because of congestion in the outgoing probe traffic. Although we do not have a hypothesis as to what mechanism could be possibly causing this behaviour, the evidence against it occurring before or during XDP execution is stronger. The brief experiment we performed on a personal laptop points to this being VM specific behaviour. To use our approach on a virtual machine we suggest further research into how it performs on different virtualization implementations.

5.3 Filter

We have stated before that both the throughput and droprate statistics are not significantly impacted by the choice of IP filter. The overall result from these being that the bitmap filter slightly outperforms the Bloom filter or that no difference can be observed. However, in section 4.4 we observed that the Bloom filter takes about an order of magnitude longer than the bitmap filter to be constructed. From these results we see no advantage of using the Bloom filter over the bitmap filter.

5.4 Usability

The components of our approach are ZMap, XDP, the loopback interface and a rust executable to load the XDP code and create the filters. XDP is included by default in kernel version 4.8 and newer. Thus it is possible to run it on any Linux machine, including virtual machines, with a kernel 4.8+ and a rust compiler and ZMap installed, or any other mass scan tool. This means that it is also possible to run it on a VM on top of, for instance, a windows operating system.

In practice we have verified our approach to function on Debian with kernel version 5.10, with at least 16GB of RAM, including a virtual machine running on KVM. However, the droprate on our VM test case increases with the send rate and might make it less suitable to be run at high send rates on virtual machines. Tests with different hypervisors will have to definitively show this.

5.5 Future Work

5.5.1 Better performance

Any attempt to speed up our approach that would not address speeding up ZMap will be limited by the baseline throughput measured. Since we measured the maximal speed that ZMap is able to achieve on our test machines, going beyond this with our approach would require the sending of the packets to be faster. Significant speed up would thus not come from modifying the XDP part of our approach but from modifying the packet generation part. For instance tests could be performed with different mass scan tools to see if these are able to reach a higher send rate. It might also be worthwhile to investigate the possibility of keeping the full execution is user space. ZMap runs in user space and XDP in kernel space, requiring context switches to process the probes. If a different approach would intercept the probes before being handed to the kernel then these context switches could be prevented.

Furthermore a mass scan tool with zero-copy to prevent the copying of the packets to kernel space could provide speed up. For instance, investigating the use of AF_XDP with a mass scan tool would be appropriate.

5.5.2 Investigate droprate

The high droprate observed in the VM tests significantly affects the usability of this tool on the VM we have tested. An informal test points towards the VM environment being the cause of this behaviour, however further investigation has to show whether this is the case. Investigation into whether this behaviour also occurs on other test machines with different kernels, hypervisors or virtualization approaches is important to further evaluate the usability of our approach on virtual environments. Furthermore attempting to reproduce this behaviour on different kernels, hypervisors or hardware could point towards the source of this problem. Besides this, an investigation into the networking of the KVM virtual machine, would be useful to finding out why this behaviour is occurring.

Chapter 6

Conclusion

In this research we acknowledged and argued the need for a sandbox environment for mass scan tools in an educational and research context. To contribute towards fulfilling this need we designed and created a number of prototypes. Of these we selected the final one to be benchmarked and evaluated. We presented the results of this evaluation and discussed their implications on the requirements and goals set out. In this chapter we will conclude what this means for the research question.

In the introduction we asked the research question *"How to make a virtual responder that can intercept, filter and respond to probes at the scale and speed of mass scan tools?"*

We can formulate an answer to this question using this research: using the approach described in section 3.1 a virtual responder can be made which intercepts filters and responds to probes at the scale and speed of mass scan tools. As we have shown this can be achieved with a relatively small amount of XDP code. The XDP hook can be installed on the loopback interface to intercept the traffic and send responses back to the mass scan tool. The Bloom filter and the bitmap implemented in XDP both fulfill the function of filtering the incoming traffic, however the bitmap implementation outperforms the Bloom filter on all accounts except for filter size. The use of the loopback interface means that this approach is entirely virtual and no traffic leaves the machine that is used. Our results show that our approach works at the scale of a full IPv4 scan, and we do not expect a technical limitation to use it at the scale of partial IPv6 scans. The bitmap filter, however, does not scale beyond the full IPv4 scan, but the Bloom filter does. Furthermore our results show that our approach is able to run at close to the speed of the mass scan tools. The biggest slowdown being a 12 minute penalty on an unburdened speed of 1 hour for a full IPv4 scan, and the smallest being 0.7 minutes on an unburdened speed of 27.7 minutes. Regretfully we can not conclude whether the approach performs sufficiently in a VM setting. The setting we used for testing had a yet unexplained high percentage of responses being dropped before reaching the mass scan tool receiver. Further research will have to show whether this result is inherent to this approach in a virtualized environment or whether it is an issue with the specific testbed. Regardless we can conclude that ReScan functions as a virtual responder at the scale and speed of mass scan tools.

Bibliography

[1] "ZMap: The Internet Scanner," Dec. 2022, original-date: 2013-01-23T01:30:09Z. [Online]. Available: https://github.com/zmap/zmap

[2] R. D. Graham, "MASSCAN: Mass IP port scanner," Dec. 2022, original-date: 2013-07-28T05:35:33Z. [Online]. Available: https://github.com/robertdavidgraham/masscan

[3] "VulnerabilityAssessment.co.uk." [Online]. Available: http://www.vulnerabilityassessment.co.uk/scanrand.htm

[4] "unicornscan | Kali Linux Tools." [Online]. Available: https://www.kali.org/tools/unicornscan/

[5] Z. Durumeric, E. Wustrow, and J. Halderman, "ZMap: Fast internet-wide scanning and its security applications," 2013, pp. 605–619.

[6] "LIBPCAP 1.x.y by The Tcpdump Group," Dec. 2022, original-date: 2013-04-14T21:46:36Z. [Online]. Available: https://github.com/the-tcpdump-group/libpcap

[7] T. Zhang, L. Linguaglossa, M. Gallo, P. Giaccone, L. Iannone, and J. Roberts, "Comparing the performance of state-of-the-art software switches for NFV," 2019, pp. 68–81.

[8] "DPDK." [Online]. Available: https://www.dpdk.org/

[9] L. Rizzo, "NetMap: A novel framework for fast packet I/O," 2019, pp. 101–112.

[10] T. Høiland-Jørgensen, J. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The eXpress data path: Fast programmable packet processing in the operating system kernel," 2018, pp. 54–66.

[11] M. Karlsson and B. Töpel, "The path to DPDK speeds for AF XDP," in *Linux Plumbers Conference*, 2018.

[12] “Source code of include/linux/socket.h of torvalds’ linux kernel 4.18.” [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/socket.h?h=v4.18#n210

[13] M. Ruiz-Sánchez, E. Biersack, and W. Dabbous, “Survey and taxonomy of IP address lookup algorithms,” *IEEE Network*, vol. 15, no. 2, pp. 8–23, 2001.

[14] H. Song, J. Turner, and J. Lockwood, “Shape shifting tries for faster IP route lookup,” vol. 2005, 2005, pp. 358–367, iSSN: 1092-1648.

[15] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor, “Longest prefix matching using bloom filters,” *IEEE/ACM Transactions on Networking*, vol. 14, no. 2, pp. 397–409, 2006.

[16] W. Eddy, “Transmission Control Protocol (TCP),” Internet Engineering Task Force, Request for Comments RFC 9293, Aug. 2022, num Pages: 98. [Online]. Available: https://datatracker.ietf.org/doc/rfc9293

[17] A. Kirsch and M. Mitzenmacher, “Less hashing, same performance: Building a better Bloom filter,” *Random Structures & Algorithms*, vol. 33, no. 2, pp. 187–218, 2008, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/rsa.20208. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/rsa.20208

[18] Bryce Kahle, “How and When You Should Measure CPU Overhead of eBPF Programs.” [Online]. Available: https://ebpf.io/summit-2020

[19] D. Adrian, Z. Durumeric, G. Singh, and J. A. Halderman, “Zippier ZMap: Internet-Wide Scanning at 10 Gbps,” 2014. [Online]. Available: https://www.usenix.org/conference/woot14/workshop-program/presentation/adrian

Appendix A

Variance Graphs

In this appendix we collected the standard deviations of the mean from the experiments we conducted. These are also shown in the results as error bars, however, we also included it in this graph form for ease of reading.

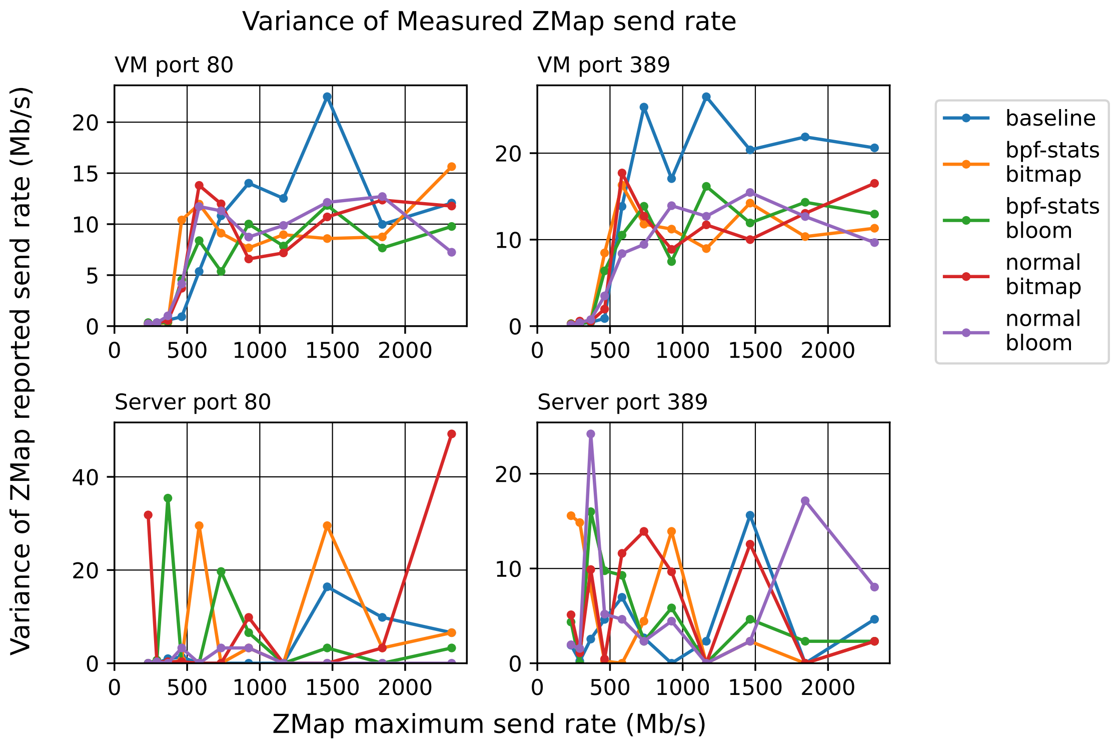


Figure A.1: Standard deviation of the rate at which ZMap reports to have sent out packets, plotted against what rate it was set to not exceed. Mean data can be found in figure 4.1

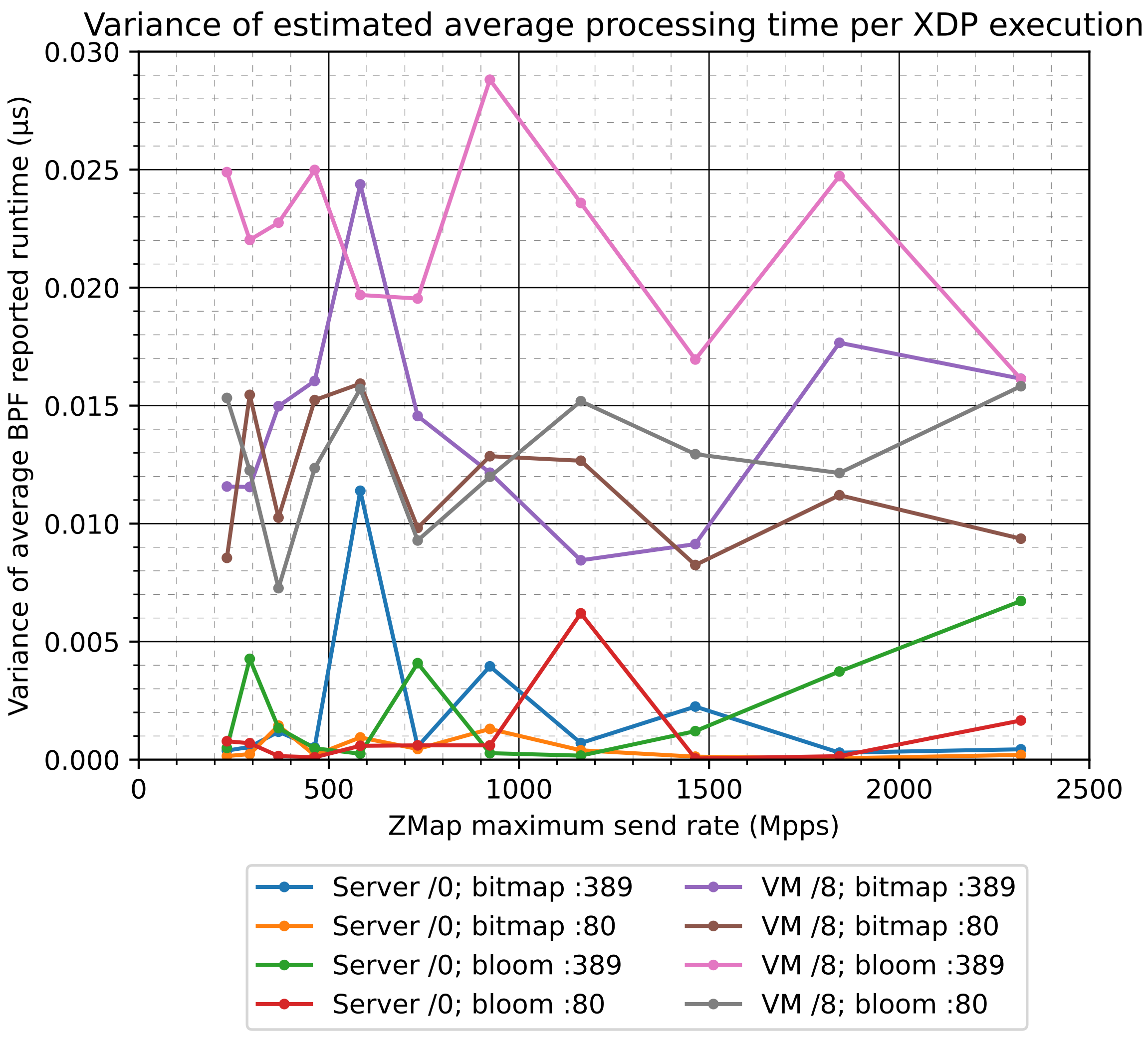
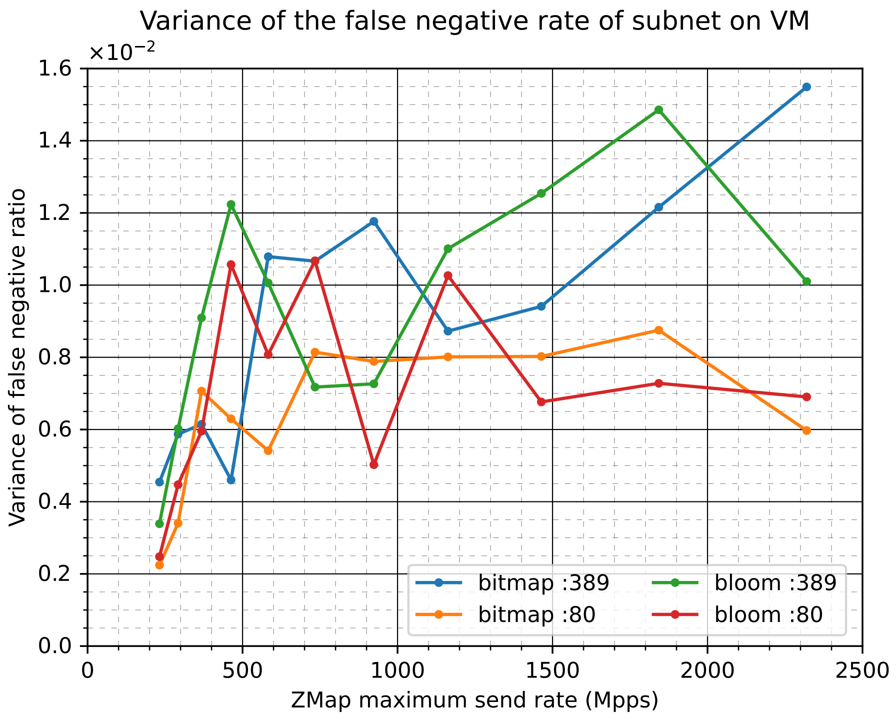


Figure A.2: Standard deviation of the estimated average time it took ReScan to process a single probe plotted against the send rate ZMap was set to not exceed. Mean data can be found in figure 4.2



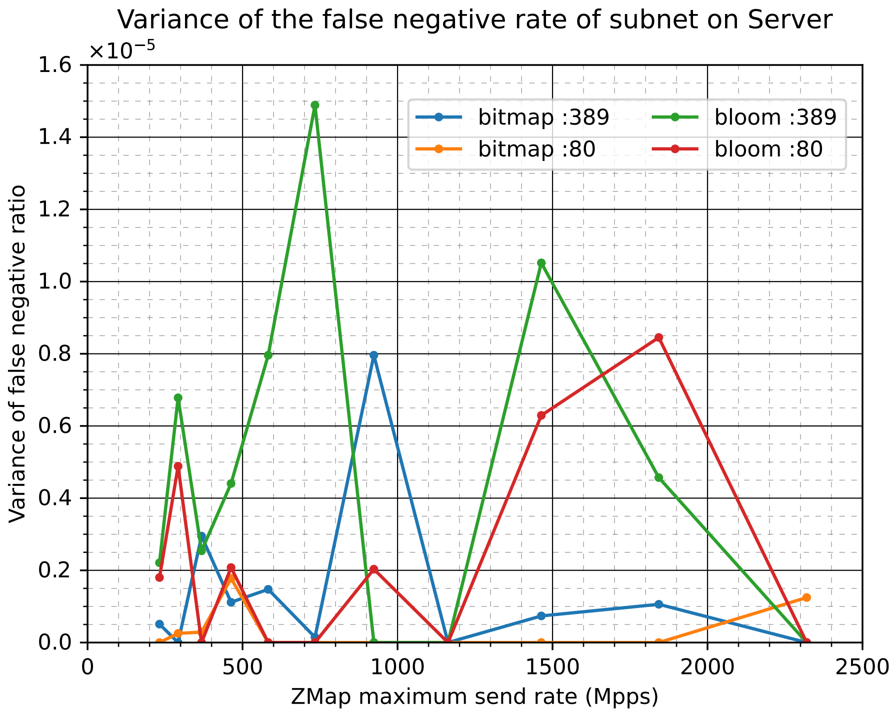


Figure A.3: Standard deviation of the amount of responses not received that should have been received,as a ratio of the amount that should have been received, on the VM and server testbed respectively. Mean data can be found in figure A.3

Appendix B

Code listings

In this appendix we provide code snippets of the essential parts of ReScan. For brevity we marginally simplified some parts of the code. e.g. replacing function calls with the function body, or removing the code supporting a potential expansion to IPv6 in the future. All logic and control flow is equivalent.

```rust
fn try_responder(ctx: XdpContext)
    -> Result<xdp_action::Type, xdp_action::Type>
{
    let mut hdr_cursor = 0usize;
    let (eth, ip) = unsafe {
        parse_routing(&ctx, &mut hdr_cursor)
            .ok_or(xdp_action::XDP_PASS)?
    };
    let protocol = unsafe { (*ip).protocol };
    let daddr = unsafe { (*ip).daddr };

    // Pass packet if it is locally addressed
    // to not disturb normal loopback traffic
    if (daddr & 0xFF000000) == 0x7F000000 {
        return Ok(xdp_action::XDP_PASS);
    }
    // Drop packet if it does not match the filter
    if unsafe { !matches_filter(&ctx, daddr) } {
        return Ok(xdp_action::XDP_DROP);
    }
    // Pass packet if it is not a TCP packet,
    // and thus not sent by mass scan tool
    if protocol != IPPROTO_TCP { return Ok(xdp_action::XDP_PASS); }

    let tcp = parse_tcphdr(&ctx, &mut hdr_cursor)
        .ok_or(xdp_action::XDP_PASS)?;
    let tcp_syn = unsafe { (*tcp).syn() };
    let tcp_ack = unsafe { (*tcp).ack() };
    // Pass packet unless it is a SYN packet but not an ACK packet
    if tcp_syn == 0 || tcp_ack != 0 { return Ok(xdp_action::XDP_PASS); }

    // Otherwise 'bounce' packet: rewrite the ETH,IP and TCP
    // headers to be a valid response
    unsafe {
        bounce_eth(&ctx,eth);
        bounce_ip(&ct,ip);
        bounce_tcp(&ctx, tcp);
    }
    return Ok(xdp_action::XDP_PASS)
}
```

Listing B.1: The main packet-processing code

```rust
const SPOOF_SOURCE_MAC: [u8; 6] = [0xFE, 0, 0, 0, 0, 0];
#[inline(always)]
unsafe fn bounce_eth(_ctx: &XdpContext, eth: *mut ethhdr) {
    (*eth).h_dest = (*eth).h_source;
    // Use a MAC from the Locally Administered Address Ranges
    // to prevent mass scan tool believing it sent the reply
    // itself
    (*eth).h_source = SPOOF_SOURCE_MAC;
}
```

Listing B.2: 'Bounce ethernet': rewriting ethernet header to be a response

```rust
#[inline(always)]
unsafe fn bounce_ip(_ctx: &XdpContext, ip: *mut iphdr) {
    // IP checksum not changed because of swap
    mem::swap(&mut (*ip).daddr, &mut (*ip).saddr);
}
```

Listing B.3: 'Bounce IP': rewriting IP header to be a response

```rust
#[inline(always)]
unsafe fn bounce_tcp(_ctx: &XdpContext, tcp: *mut tcphdr) {
    // Swap source/destination port and ack/seq number
    // to keep checksum the same as much as possible
    mem::swap(&mut (*tcp).source, &mut (*tcp).dest);
    mem::swap(&mut (*tcp).ack_seq, &mut (*tcp).seq);

    // If overflow: 1's complement sum is unchanged
    let (ack_seq, o) = u32::from_be((*tcp).ack_seq)
        .overflowing_add(1);
    (*tcp).ack_seq = u32::to_be(ack_seq);
    (*tcp).set_ack(1);

    // Add delta of acknowlegment number (if no overflow)
    // and the ack flag to the checksum
    (*tcp).check = !(ones_complement_add_u16(
        !(*tcp).check,
        (!o as u16) + (1 << 4))
    );
}
```

Listing B.4: 'Bounce TCP': rewriting TCP header to be a response

```rust
// The filter consists of a BPF_ARRAY_MAP,
// with entries being 'chunks' of type filter::ChunkType,
// which is a rust array of words of type filter::WordType,
// which is by default a byte, encoding 8 boolean values
// This word is indexed most significant -> least signifant
#[map(name = "FILTER_MAP")]
static FILTER_MAP: Array<filter::ChunkType> =
    Array::<filter::ChunkType>
         ::with_max_entries(filter::MAP_SIZE as u32, 0);

#[inline(always)]
unsafe fn matches_filter(_ctx: &XdpContext, daddr: IpAddr)
    -> bool
{
    // apply the ADDRESS_MASK: only keep offset from subnet
    // of this filter
    let key = daddr & filter::ADDRESS_MASK as u32;

    // Split the key into a map index pointing to a chunk
    let map_i = key >> (filter::ADDRESS_BITS_CHUNK as u32);
    // and split the key into a chunk index pointing to a bit
    let chunk_i = key & (filter::ADDRESS_MASK_CHUNK as u32);

    // Retrieve chunk from the filter map
    if let Some(chunk) = FILTER_MAP.get(map_i as u32) {
        // Split chunk index into index pointing to a word
        let chunk_i = (chunk_i as usize)
            >> filter::ADDRESS_BITS_WORD;
        // and split chunk index into a word index pointing to the bit
        let word_i = chunk_i & (filter::ADDRESS_MASK_WORD as u32);

        // Retrieve word from chunk
        let word = chunk[chunk_i];
        // Retrieve bit from word(byte)
        return (word >> (filter::ADDRESS_MASK_WORD as u32 - word_i))
                & 1 == 1
    } else {
        // Should never happen
        return false
    };
}
```

Listing B.5: Bitmap filter matching

```
// The filter consists of a BPF_ARRAY_MAP,
// with entries being 'chunks' of type filter::ChunkType,
// which is a rust array of words of type filter::WordType,
// which is by default a byte, encoding 8 boolean values
// This word is indexed most significant -> least signifant
#[map(name = "FILTER_MAP")]
static FILTER_MAP: Array<filter::ChunkType> =
    Array::<filter::ChunkType>
        ::with_max_entries(filter::MAP_SIZE as u32, 0);

#[inline(always)]
unsafe fn matches_filter(_ctx: &XdpContext, daddr: IpAddr) -> bool {
    // If test is true for all k hash values, assume daddr is in set
    for hash_offset in 0..bloom_filter::HASH_COUNT {
        // Compute hash value
        let hash = bloom_filter::hash(daddr, hash_offset);
        // Split the hash key into a map index pointing to a chunk
        let map_i = hash >> (filter::ADDRESS_BITS_CHUNK as u32);
        // and split the key into a chunk index pointing to a bit
        let chunk_i = hash & (filter::ADDRESS_MASK_CHUNK as u32);
        // Retrieve chunk from the filter map
        let test = if let Some(b) = FILTER_MAP.get(map_i as u32) {
            // Split chunk index into index pointing to a word
            let word_i = chunk_i & (filter::ADDRESS_MASK_WORD as u32);
            // and split chunk index into
            // a word index pointing to the bit
            let chunk_i = (chunk_i as usize)
                >> filter::ADDRESS_BITS_WORD;
            // Retrieve word from chunk and bit from word(byte);
            // interpret as boolean value
            (b[chunk_i] >> (filter::ADDRESS_MASK_WORD as u32 - word_i))
                & 1 == 1
        } else {
            // Should never happen
            false
        };
        // If test is false for 1 hash value,
        // we know daddr is not in set
        if !test {
            return false
        }
    }
    return true
    }
```

Listing B.6: Bloom filter matching

```rust
// Hashing function based on jhash.h from the linux kernel,
// by Bob Jenkins and Jozsef Kadlecsik in Public Domain
const HASH_INITVAL: u32 = 0xdeadbeef;

#[inline(always)]
pub fn hash(key: u32, initial_value: u32) -> u32 {
    let mut b = HASH_INITVAL.wrapping_add(initial_value);
    let mut c = b;
    let mut a = key.wrapping_add(b);
    c ^= b;
    c = c.wrapping_sub(b.rotate_left(14));
    a ^= c;
    a = a.wrapping_sub(c.rotate_left(11));
    b ^= a;
    b = b.wrapping_sub(a.rotate_left(25));
    c ^= b;
    c = c.wrapping_sub(b.rotate_left(16));
    a ^= c;
    a = a.wrapping_sub(c.rotate_left(4));
    b ^= a;
    b = b.wrapping_sub(a.rotate_left(14));
    c ^= b;
    c = c.wrapping_sub(b.rotate_left(24));

    return c & filter::ADDRESS_MASK as u32;
}
```

Listing B.7: Hashing function

```rust
pub mod filter {
    // Full address bit-width
    // retrieved from envvar FILTER_ADDRESS_BITS
    // default = 32
    pub const ADDRESS_BITS: usize = [...]
    // Amount of bit-entries in filter
    // default = 0x100_000_000 = 512MB
    pub const BITS: usize = 1 << ADDRESS_BITS;
    // default = 0xff_fff_fff
    pub const ADDRESS_MASK: usize = BITS - 1;

    // Word address bit-width
    pub const ADDRESS_BITS_WORD: usize = 0x3;
    pub type WordType = u8;
    // Amount of bit-entries in a word = 1 << ADRESS_BITS_WORD
    pub const WORD_BITS: usize = 0x8;
    pub const ADDRESS_MASK_WORD: usize = WORD_BITS - 1;


    // Chunk address bit-width
    // retrieve from envvar FILTER_ADDRESS_BITS_CHUNKS
    // default = 18 = log_2(32KB)
    pub const ADDRESS_BITS_CHUNK: usize = [...]
    // Amount of bit-entries in a chunk
    // default = 0x40000  = 32KB (per-cpu map value size limit)
    pub const CHUNK_BITS: usize = 1 << ADDRESS_BITS_CHUNK;
    pub const CHUNK_BYTES: usize = CHUNK_BITS >> 0x3;
    // Amount of bytes in a chunk
    // WORD_SIZE * CHUNK_SIZE = 32KB
    pub const CHUNK_SIZE: usize = CHUNK_BITS >> ADDRESS_BITS_WORD;
    pub type ChunkType = [WordType; CHUNK_SIZE];
    pub const ADDRESS_MASK_CHUNK: usize = CHUNK_BITS - 1;

    // Amount of chunks
    pub const MAP_SIZE: usize = BITS >> ADDRESS_BITS_CHUNK;
}
```

Listing B.8: Filter constants