Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

MASTER'S THESIS IN INFORMATICS

Network Discovery Orchestration

Michael Eder

Technical University of Munich

Department of Informatics

Master's Thesis in Informatics

Network Discovery Orchestration

Orchestrierung von Netzwerkerkundung

Author:	Michael Eder
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Jonas Jelten, M. Sc.
	Simon Bauer, M. Sc.
Date:	April 15, 2019

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Garching, April 15, 2019

Location, Date

Signature

Abstract

Port scanning is a common approach for discovering services in networks. While advances in vertical scaling using high speed uplinks made it possible to scan large and reliable networks like the Internet, horizontal scaling by joining a fleet of scanners has been neglected in the last few years. Distributing scan tasks across workers using an agent allows to improve performance, accuracy and reliability without imposing requirements like uplink sizing and is therefore an interesting approach for scanning private, e.g. company networks. This work presents a prototype that performs distributed port and application layer scans without requiring elevated privileges. The prototype is supposed to be portable, supporting major operating systems (Windows, Linux, macOS, BSDs) on major platforms (Intel, MIPS, ARM) as static binaries without runtime dependencies. Our measurements show that the prototype is about ten times faster than Nmap using a single node and adding more scanner nodes linearly increases scan speed. Its accuracy is comparable to ZMap, Masscan and Nmap. This work shows that scan distribution allows to gather additional insights and overcome some limitations that prevent usage of high speed scanners in some real world scenarios while providing accurate results and significant scan speed improvements compared to Nmap.

Zusammenfassung

Portscans sind ein geläufiger Ansatz, um Dienste in einem Netzwerk zu entdecken. Während Fortschritte im vertikalen Skalieren durch die Nutzung von Hochgeschwindigkeitsuplinks das Scannen von großen und zuverlässigen Netzwerken wie dem Internet ermöglichten, spielte horizontales Skalieren durch Zusammenschließen einer Scannerflotte in den letzten Jahren kaum eine Rolle. Verteilen von Arbeit auf mehrere Scanner ermöglicht Verbesserungen hinsichtlich Leistung, Genauigkeit und Zuverlässigkeit ohne Anforderungen wie Uplinkdimensionierung zu stellen und ist daher ein interessanter Ansatz um private Netzwerke, beispielsweise in Unternehmen, zu scannen. Diese Arbeit präsentiert einen Prototypen, der in der Lage ist, einen verteilten Port- und Anwendungsschichtscan durchzuführen, ohne dafür erhöhte Berechtigungen zu erfordern. Der Prototyp ist portabel und unterstützt die wichtigsten Betriebssysteme (Windows, Linux, macOS, BSDs) auf den wichtigsten Plattformen (Intel, MIPS, ARM) als statisches Binary ohne Laufzeitabhängigkeiten. Unsere Messungen zeigen, dass der Prototyp mit einem einzelnen Knoten rund zehnmal so schnell wie Nmap ist. Das Hinzufügen von weiteren Knoten beschleunigt die Scangeschwindigkeit linear. Die Genauigkeit des Prototypen ist vergleichbar mit ZMap, Masscan und Nmap. Diese Arbeit zeigt, dass Scanverteilung zusätzliche Einblicke ermöglicht und manche Beschränkungen, die den Einsatz von Hochgeschwindigkeitsscannern in realen Szenarien oftmals verhindern, umgehen kann. Desweiteren sind die Ergebnisse akkurat und die Scangeschwindigkeit kann im Vergleich zu Nmap signifikant erhöht werden.

I would like to thank Prof. Dr.-Ing. Georg Carle for giving me the opportunity to write this thesis at the Chair of Network Architectures and Services. This work would not have been possible without Jonas Jelten and Simon Bauer that empowered me to work on this subject, gave me a lot of valuable feedback from the very beginning until the end and pointed me into right directions. Special thanks go to the Rechnerbetriebsgruppe (RBG), preparing the ground for and allowing me to perform various scans against their network. Manuel Burghard, Lukas Heinzmann and Daniel Kowatsch gave me valuable feedback on software architecture. I like to thank all proof-readers and the authors of the free and open source software I used for their great work, too. Last but not least I would like to thank my family, especially my grandparents, for giving me the opportunity to study computer science at TUM and supporting me in everything I am doing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Network discovery is the process of finding open ports and services in a network, usually by using a port scanner that actively sends packets to hosts in order to determine which ports are open and which services are listening. This is useful to get an understanding of services available on a network, identifying vulnerabilities and weak spots as well as uncovering misconfigurations. Therefore, port scanners are used by the curious, security researchers, penetration testers, during audits and on any occasion where it is required to find out which services are provided by which machines on a network.

Official releases of existing scanners come with environment requirements for their operation. This means that they either have to be installed, require special operating system libraries, root permissions or are not available for mainstream operating systems or architectures. Especially when scan orchestration across multiple hosts in non-Internet environments is desired, these limitations often become difficult to deal with. For example, it is sometimes hard to get scanning infrastructure deployed into networks of interest (e.g. DMZs, IoT networks etc.). Runtime requirements of existing scanners do not allow easy, non-intrusive deployment, for example in common corporate environments running mostly Microsoft Windows with unprivileged user accounts assigned to most employees.

Existing tooling lacks capabilities regarding setup, execution and result collection in such scenarios.

1.2 Research questions

In order to distribute a network scan across different nodes, some problems have to be addressed. Scanning at high speed stresses network infrastructure and may disrupt normal operation, therefore it is required to find a way to throttle scanner nodes without having access to operating system functionality that requires elevated privileges. Next, a method needs to be devised to split scan targets into work batches that can be distributed. How is it possible to make sure that each target is scanned exactly once by each node and how to handle detail problems like work reassignment in churn situations? Also, are there any difficulties regarding result collection and merging? It is important that no data inconsistencies arise in failure situations and results should be ingested and processed easily by a wide variety of tools, e.g. database systems or visualization software. It has to be possible to stop any scan across all nodes before it is finished, for example when the scan interferes with normal network operation. What is a reliable way to implement stopping the scan in this distributed scenario and how can a dead man's switch, stopping nodes in case of server failure, be realized?

The goal of this work is to implement a prototype that is going to address these questions and problems. It is able to automatically distribute work across scanner nodes while being portable across architectures and operating systems and running at a low privilege level.

1.3 Outline

The remainder of this work is structured as follows. Chapter 2 gives background on the topic of port scanning by introducing existing scanners and previous work that dealt with the distribution of scan tasks. Chapter 3 describes the requirements and the derived goals for the prototype. Chapter 4 gives an overview over the scanner architecture as well as the concrete implementation and practical problems that were faced. Chapter 5 contains evaluations in a testbed as well as data collected by performing various real world scans. It also contains comparisons with other well known tools. Chapter 6 concludes this work, gives an outlook and provides ideas for improving the prototype in the future.

Chapter 2

Related work and state of the art

2.1 Port scanning

As scanning ports has been an interesting problem since the early days of the Internet, there are various port scanning approaches and several scanners implementing various feature sets available. The following chapter is going to give an overview and describes the state of the art. Further comparisons, especially when it comes to speed between synchronous and asynchronous approaches, can be found in [8].

2.1.1 Nmap

Nmap [17] is a well-known and widely used port scanner written in C. It was initially released in September 1997 [11] and is still under active development. Nmap uses raw IP packets if possible in order to find out which hosts, services, operating systems and packet filters are in use. It is platform independent and there is a graphical user interface as well as supporting tooling like a diff tool, packet generators, and a lot of unofficial tooling that e.g. allows to visualize the scan data. Nmap is free software and has detailed documentation as well as extensive configuration options. A lot of different scan types are supported, e.g. TCP Connect and TCP SYN scans or UDP probing, but also uncommon scans like the XMAS scan which enables uncommon flag combinations in TCP headers to trigger responses.

Nmap comes with service and operating system detection capabilities. In order to find out which software is running on a remote location, Nmap tries to detect nuances in protocol implementations like TCP or has predefined payloads that are sent in order to hopefully trigger responses containing relevant information, e.g. responses containing HTTP server headers. A powerful scripting engine based on the Lua scripting language is supported and Nmap comes with lots of scripts bundled which perform higher-level analysis, there is for example a script that prints exported mount points in case of open NFS servers.

Nmap tries to be as accurate as possible, which imposes some problems. First, it keeps the probe state for timing and proper retransmissions which makes it slow when scanning large networks or large port ranges. Second, the timing algorithm assumes congestion in case of packet loss. In cases where probes are dropped by packet filters or hosts are non existent, Nmap may throttle probing, therefore scans of large networks or port ranges with sparse replying services can heavily influence the performance of a Nmap scan [24].

2.1.2 Asynchronous scanners

Popular alternatives among researchers are asynchronous scanners that allow the utilization of links close to their line speed, but due to architectural considerations results produced by these scanners are susceptible to false negatives in case of packet loss, e.g. due to congestion or overloading of target hosts or networks. These scanners are usually used to perform Internet wide scans because it is possible to fully utilize well-dimensioned uplinks, while equal distribution of concurrently scanned targets across a large network minimizes the risk of packet loss. Running these scanners against single hosts or small networks with single hot spots like switches or routers can lead to insufficient results while simultaneously risking the interruption of normal network operation.

ZMap and related tools

ZMap [8] is an open source scanner released in 2013 and written in C. Since Nmap is too slow for scanning for open ports at Internet scale, ZMap started to employ an approach that asynchronously sends probes and decides, based on a mechanism similar to SYN-Cookies, if a response belongs to a probe sent earlier. This way, it was possible to utilize a gigabit uplink for a scan of a single port across the Internet and later improvements allowed to utilize a 10 gigabit uplink [2].

To ease working with results or gaining additional insights, the ZMap project also released additional utilities, e.g. ZTee, ZGrab, ZTag et al. and described how these tools are combined to realize a search engine that allows to query the data set of services exposed to the Internet [9].

ZMap was built for scanning a single port across the Internet. It assumes that the scanning host has sufficient upstream bandwidth and it is not possible to perform a scan using multiple target ports. Also, ZMap is not platform independent as it only supports unix-like operating systems like Linux, macOS or BSD. Since packets are sent at high speed as raw Ethernet frames, it is required to have elevated privileges as well as correct hardware, drivers and libraries available on the system.

The ZMap project runs Internet-wide scans regularly and makes the result accessible via a search engine [4]. The project also hosts a repository of Internet scan data[1].

Masscan

Masscan [14] by Robert David Graham was released in 2013, is open source software that is written in C and has a lot of similarities to ZMap. It has its own TCP/IP stack and sends packets asynchronously, claiming to be able to reach 10Gb/s speed. Elevated privileges are required in order to perform a scan using Masscan. In contrast to ZMap, it allows arbitrary port ranges to be scanned in a single run and it is possible to build and operate Masscan on Linux, Windows, macOS and FreeBSD. For selected protocols application layer scans and vulnerability checks are implemented, e.g. HTTP banner grabbing, SMB enumeration, SSL 3 checks or Heartbleed vulnerability detection.

2.1.3 Distributed scanning approaches

To overcome limitations such as single scanner speed, it is possible to distribute a port scan across different nodes. The scan may be faster due to the reduced number of targets that need to be scanned by each node or results may be more accurate, depending on node placement inside the network and how many nodes are scanning each target. For example, a scan distributed across nodes in different network segments may reveal firewall misconfigurations. There are few maintained frameworks for scan distribution and most of the frameworks act as wrappers around existing port scanners.

In order to distribute a port scan, it is important to split the targets into subsets that are scanned by nodes. The target generation algorithms in ZMap and Masscan allow the generation of unique, equally sized, disjunct subsets of target networks. This means that multiple instances can scan equally small, separate parts of a target network. Unfortunately, Masscan and ZMap do not implement further distribution or collection mechanisms, but the general ability to work on network subsets make them interesting scanner engines for static deployments of scanner fleets.

Dnmap

Dnmap [7] is a Python wrapper that connects Nmap instances, but it has not been maintained since 2013. Newer approaches like Scantron [1] have additional features like

[1] `https://scans.io/`

a web interface for controlling scanner nodes as well as data collection and visualization. Scantron uses SSH, NFS and PostgreSQL for communication, data sharing and analysis. The setup is a static approach, using Ansible for deployment on Ubuntu installations.

Unicornscan

Unicornscan [38], a port scanner that has not been maintained since 2009 due to the death of its author [36], allowed rudimentary distribution of a scan across several *Drones*. To use this functionality, the client is started in drone mode and once all drones are set up, the master simply connects to drones via TCP instead of forking local workers [18]. There is no drone authentication. The GitHub mirror of the project [37] contains several warnings that the code is not finished and may not be safe to use. Unicornscan runs on unixoide operating systems.

Wolpertinger

Wolpertinger [39] is a distributed port scanner that has sender and listener drones [40]. Sender drones send TCP packets containing the spoofed sender IP of the listener drone to targets and in case of an open port, the listener drone receives the reply. There is challenge-response authentication for drones and Wolpertinger allows to handle drone failures. Wolpertinger runs on Linux, was released in 2012 and received a commit in 2017, but there has been no public development activity or clues of people using it ever since and the project seems to be dead.

Distributed scanning of application layer protocols

In addition to port scans, information on other protocols might be a subject of interest. Van Rijswijk-Deij et al. [28] describe their infrastructure setup to perform active DNS measurements. They also rely on a static setup that is manually configured beforehand.

All scanners and services named earlier provide additional features like service scans or access to service scan data, e.g. DNS, available TLS ciphers and certificates presented, HTTP banner grabbing etc. For example, Nmap can be instructed to attempt to find out which known services and versions are running by grabbing banners.

2.2 Internet wide scans and scanning services

There are several projects regularly performing Internet wide scans. RIPE Atlas continuously measures “Internet connectivity and reachability, providing an unprecedented understanding of the state of the Internet in real time“ [29] and the project claims to build the largest measurement network in history. Services like Shodan [30] and Censys [4], the latter one built and operated by the ZMap authors, make their scan data publicly available via a web interface and offer payed subscriptions for customers with additional features like API access or extended functionality. That way it becomes easy to find out which services are distributed over the Internet and it becomes possible to evaluate the impact of new security vulnerabilities because estimating numbers of affected devices connected to the Internet does not require a full scan anymore.

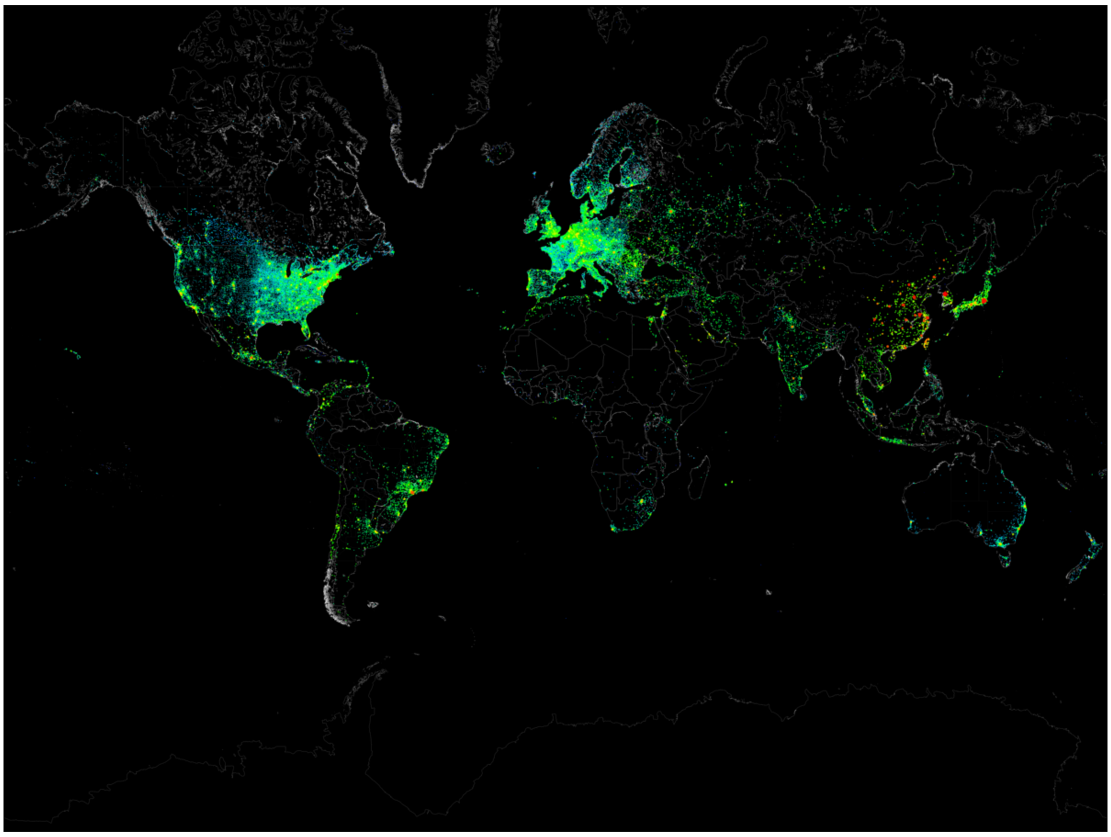

Figure 2.1: Visualization of 460 Million IP addresses and their geographic location between June 2012 and October 2012. The data was collected by the Carna Botnet [16].

Some services allow to download the data sets, e.g. Project Sonar [26] and Censys. This allows to perform detailed offline analysis of these large data sets. It is also possible to derive higher-level metrics, like Security Research Labs' hackability score across countries and industries [15].

The first publicly known, systematic scan of the public IPv4 address range was done in 2012 by an anonymous researcher in a project called *Internet Census 2012* [16]. The Internet was scanned for devices with default `telnet` login credentials and every time vulnerable systems were found, a binary was uploaded and executed. It started another instance scanning for insecure telnet devices, again uploading the binary whenever a target was identified. The resulting botnet was named *Carna* and consisted of approximately 420.000 devices that performed port scans. The data was collected and analyzed and the results were the first to give interesting insights into the state of the Internet regarding aspects like port and service distribution or correlation between local time and connected devices. Figure 2.1 shows one of several visualizations created using the collected data.

Currently there are several tools, services and also businesses that allow to generate, access or process port scan data, but unfortunately there is no freely available tool that fulfills our requirements. These, as well as the goals for the prototype presented in this work, are explained in the next chapter.

Chapter 3

Distributed Network Discovery

As already described in chapter 1.1, there are scenarios that prevent usage of existing scanners or dedicated scanning infrastructure. Combined with the lack of tools that have built in scan distribution capabilities, an approach was taken to overcome most of these problems. The following sections describe requirements and goals of the prototype to make port scanning under constraints easier and more flexible. Chapter 4 explains the concrete architecture and implementation of the prototype.

3.1 Requirements

In order to decide what to focus on and what to ponder in case of design decisions and conflicting goals, all requirements for the prototype were defined upfront.

The prototype should be *reasonably* fast. This does not mean that it should be faster than all other scanners, which is particularly hard when considering asynchronous scanners, but a single node should have at least scan performance comparable to other stateful scanners.

Next, distribution of nodes has to be possible in an easy and flexible way. This also implies that nodes are supposed to run on as many operating systems and architectures as possible without special requirements like drivers, operating system libraries or elevated user permissions. Nodes should leave no configuration or file system artifacts apart from their binary, therefore stopping the node and deleting the binary have to be sufficient to remediate the system. Distribution of work across scanner nodes serves three purposes: Speeding up the scan, improving result quality due to generation of different target views and improved resilience against scanning nodes shutting down.

The generated data must be available while a scan is running and human readable as well as automatically processable.

Finding open ports is the first step to perform a deeper analysis of higher layer protocols. Integration of on demand scans of well known protocols like HTTP, SSH or TLS should be possible. In case of UDP scanning, the payloads used for probes are supposed to be configurable by the user.

There are also properties that are explicitly *not* a requirement. When discussing port scanning and port scanners, many views are opinionated towards scan speed as the most important feature. Because there are already fast, asynchronous scanners available that are successfully used for Internet wide surveys day in, day out, the goal for the prototype is to be fast compared to stateful scanners as long as it does not interfere with other goals.

Scanners like Nmap saw decades of development, resulting in an impressive feature set and protocol support. The prototype is supposed to focus on the task of straightforward discovery of open TCP and UDP ports using operating system interfaces as well as superficial analysis of selected higher layer protocols. Loosely related tasks such as ping scans, traceroutes, vulnerability checks or operating system detection are of little interest for this work and therefore not considered relevant features.

3.2 Goals

As seen in chapter 2, there are several excellent free tools mostly doing a good job in network discovery. We tried to identify pain points in common scenarios which together are addressed by none of the existing tools in a sufficient way. Together with the requirements defined earlier, they are used to form the goals that are going to be implemented by the prototype.

3.2.1 Performance

Performance is an important factor for network discovery. The Internet has grown to a size that is not addressable via IPv4 anymore [10] and many organizations are running private networks with tens of thousands of hosts inside. The development of asynchronous scanners allowed researchers to catch up and efficiently enumerate services across the Internet. Unfortunately, many networks are less accessible for placing dedicated scanning infrastructure, especially organization-internal networks. Often these networks are fragile and do – due to closed access – usually not deal with unexpected high loads or unusual traffic. Scanning such networks at high transmission rates may negatively influence the network. The prototype should be fast enough to reliably scan a /16 IPv4 network with a list of commonly used ports without any special requirements to the underlying network, hardware, operating system or user permissions on a single work day. Additionally, in case higher scanning speed is required, the prototype should be able to scale horizontally. This means that instead of increasing scan speed by scaling up packet rates on a single scanning node, scanning nodes that collaboratively perform the scan are deployed across the network. In theory, this should allow for linear scaling without introducing new requirements like administrative access, special hardware or well-dimensioned network infrastructure connected to a single scanning host. Many large organizations maintain subsidiaries across the world that are usually connected via site-to-site VPNs. The distributed approach allows the placement of scanning nodes in physical proximity to scanned hosts and to reduce traffic sent over large distances and many hops. This may increase scan speed and result quality.

Efforts were focused on improving scans of internal networks of organizations that may not be equipped with gigabit uplinks or have single hot spots like central routers, switches or VPN gateways. As such, the primary goal regarding performance is for each scanning node to be noticeably faster than Nmap in most cases in order to be useful for e.g. penetration testers.

3.2.2 Flexibility

Most tooling doing network discovery is not flexible in a sense that prerequisites have to be met for using these tools. For example, special libraries like libpcap need to be available on the system, tools may demand installation in order to meet specific environment requirements to be fully functional, root access may be required to get low level hardware access or tools may not work on particular operating systems or architectures. In order to perform network discovery in organizations, it is often necessary to adapt to the existing infrastructure without being too intrusive. This means that connecting additional hardware is often impossible and provided hardware has to be used. Access to certain networks may be physically restricted and only possible by using restricted user accounts on already provisioned jump hosts. The prototype has to be capable of overcoming these issues that are arising in the field.

The goal is to have a single standalone binary that does not require special permissions for its operation. This allows adaption to nearly any infrastructure. Scanning nodes may for example be placed on IoT devices having sufficient resources, IP phones, printers, personal computers, hardware servers, virtualized infrastructure or even mainframes. The ability to utilize lots of existing infrastructure also introduces new possibilities to improve performance and accuracy by horizontal scaling.

3.2.3 Network views

Apart from discovering services on the network, port and service scanning is used to check if network controls are enforced and working correctly. For example, it must not be possible to access development infrastructure from guest WiFi and office users should not be able to connect to administrative interfaces of industrial control systems. Depending on network and organizational structure, obtaining and understanding the configurations of all involved systems like firewalls and routers becomes resource intensive and error prone.

The prototype is supposed to solve this problem in a pragmatic and efficient manner by giving the opportunity to create views of a network. Because the prototype is designed to be environment agnostic, it should be possible to place scanning nodes in different network segments without great effort. A scan of the complete network should yield different results for each scanning node, therefore showing which services are reachable from which vantage points inside a network.

3.2.4 Result data

In order to access result data during a scan without the risk of accessing partial data, result data should be event based, meaning that every time an open port is discovered, an event is generated. If a node is done with its work, all generated events are collected and submitted to the server. Each event is going to carry metadata like a timestamp or name and ID of the node that generated the event. This is supposed to allow tracing which scanner node discovered an open port at what time and can for example be used to find out whether a scan is the origin of a misbehaving application[1]. Having a stream of events with timestamps also allows to work with results while the scan is still running or to analyze and compare regular scans by differentiating scans depending on their timestamps.

The implementation of these requirements proofed to be useful during one of the triage scans for the evaluation in Chapter 5, when one of the scanning systems got blocked by an IDS/IPS system. It was possible to detect this within minutes during the scan because we noted that no events with open ports were generated anymore. Investigations in the logs of the IDS/IPS were eased by the fact that we were able to give a relatively precise point in time when we got blocked by providing the timestamp of the last event where an open port was discovered.

[1] For example, there are situations where port scans may cause printers to start printing nonsense.

The whole event system should be able to feed larger data analytics systems like elasticsearch[1], Splunk[2] or PostgreSQL[3] without the need of spending much time in data preprocessing. Nevertheless, a human must be able to read and understand the data. The generated data should be useful to build similar systems like Censys [4] or Shodan [30], but also for local networks.

3.2.5 Application layer scanning

Plain port scanning does not provide further insight than stating whether a given port on a given host is open or not. The interest usually lies in the application listening on the open port, therefore a scanner that is able to gain a basic set of information about common protocols on common ports is more useful. To keep all portability, performance and data format requirements there are currently no plans of having a scripting interface where scan scripts can be implemented in languages like Lua or JavaScript. Instead, an application layer scanning engine developed and used by the authors of ZMap, will be integrated and will provide higher layer scanning capabilities. As proof of concept, the prototype should support important protocols like SSH and HTTP(S). Others are subject to future work since specific glue code is required for each module.

3.2.6 Configuration

In contrast to other scanners, our prototype should be configured solely via a configuration file in YAML[4] format. This prevents the need for wrapper scripts and options don't need to be looked up or are forgotten when calling the scanner. Furthermore, a configuration file eradicates shell issues like quoting and escaping. Another big advantage is that configuration files allow to annotate configurations, e.g. why an option is set, and allow to quickly comment configurations in/out. Configuration files can easily be committed to version control systems and are easy to share. All configuration should be done at the server and should be distributed across scanning nodes.

3.2.7 Public availability

Our prototype is going to be released as free and open source software. Everybody will be allowed to use and modify it.

[1] `https://www.elastic.co/`

[2] `https://www.splunk.com/`

[3] `https://www.postgresql.org/`

[4] `https://yaml.org/`

Chapter 4

Prototype

The upcoming sections describe the architecture and implementation of most relevant parts of the prototype.

The prototype is completely written in Go [34], a programming language developed by Google. Care was taken to not use any dependencies written in other languages like C in order to keep maximum portability. This allows to build native binaries for any architecture and operating system or runtime supported by the Go compiler [12]. Go allows cross-compiling, therefore it is easy to create required binaries ad hoc on another system. These binaries are completely static and have no requirements apart from the correct architecture and operating system.

Additionally, the prototype does not require any special permission on the systems it is supposed to run on. Any user context that is at least capable of running a browser or mail client is sufficient since user space operating system APIs are used for performing the scans.

After stopping and removing the binary, there are no remaining artifacts of the scanner and no further cleanup or infrastructure restaging is required.

4.1 Design decisions

Before going into architecture and implementation, this section describes important design decisions.

Server-Client architecture

The prototype uses a standard server client architecture. The *Server*'s task is to manage the scan and assign work to *Nodes*. Nodes are receiving work, performing a scan and reporting their results back to the server.

Communication

The server communicates with nodes via TCP and a binary message format that is unencrypted. For use in hostile environments, it is possible to use TLS 1.2 secured connections where the server is authenticated and, if desired, there is also support for TLS mutual authentication using client certificates on every node.

Blacklisting in a distributed scenario

There are often subnets or hosts that must not be scanned. In this case, these targets can be blacklisted and they are skipped during the scan. The prototype is capable of blacklisting networks as well as fully qualified domain names (FQDN). Single IP addresses are simply stored as /32 networks. The blacklist is kept and applied during target generation on the server side in order to prevent sending out work packages containing targets that must not be scanned. Because DNS resolution of scan targets happens on the node, there is a side effect to consider. The server is not able to prevent a blacklisted IP from being scanned when a domain name pointing to this IP address is submitted for scanning. Therefore IPs and domain names should not be mixed for target and blacklist definitions if blacklisted items really must not be scanned. Future improvements to the scanner may also transfer the blacklists to nodes for another check after DNS resolution, but currently being aware of this should be enough to prevent this case to happen.

Problems in distributed systems

There are several problems to consider when designing a distributed service like consensus, leadership, concurrent data access and time. Since the prototype uses the standard server/client model and high availability of the server is no requirement, it is not necessary to deal with consensus and leadership because the server is the leader promoting the truth. In case concurrent data access happens, Go's channels [33] were used whenever possible, falling back to manual locking using mutexes in all other cases. Go's race detector [5] was used during the complete development phase, uncovering most data races right after introducing them.

It is not possible to rely on an accurate clock on systems where the node is running, especially on devices like a Raspberry Pi that is solely used for scanning and has no access to a reliable time server. To overcome this problem, the server sends the current time when a node initially appears and the node calculates the offset to its clock. This offset is added to the timestamp of every event generated by a node. This is of course no perfect solution and does not fulfill strict accuracy requirements, but is sufficient for this use case.

Work distribution

The central work distribution logic is realized using a pooling mechanism. Each target is scanned once by each pool and a pool may contain arbitrary many nodes whereas each node is assigned to exactly one pool. If multiple views of a target network need to be created, a pool is created for each view. Speeding up scans is realized by placing multiple nodes inside a single pool. Every pool has its own target generation routine, therefore nodes placed in different pools do not scan targets in the same order and different scanning speeds of pools do not affect operation of other pools. This way, it is possible to distribute work in a flexible and efficient way.

Churn

Any scanner node may go offline and appear again later. It is important to ensure that all hosts and ports are scanned and no incomplete results are returned. To achieve this, work batches containing a list of targets and ports to scan as well as a unique ID are prepared. The server keeps track of nodes and assigned work batches. If a node goes offline, the work batch is reassigned to another node and the server only accepts results to batches assigned to the respective node in order to prevent double reporting when work is reassigned. The events generated by each node are sent in a single batch, therefore no partial result data has already reached the server in case a node goes down.

Pull-based communication

One design decision that is different to all distributed scanning approaches we are aware of is the pull based approach used by the prototype. This means that scanner nodes proactively ask the server for more work instead of getting work pushed by the server. A pull based approach has the advantage that the server does not need to discover nodes which highly increases flexibility as nodes can simply come and go.

Often, it may be easier to traverse NAT boundaries and firewalls using a pull-based approach because chances are that outgoing traffic is allowed and stateful firewalls usually allow already established connections to pass. The downside of this approach together with our message queue implementation is that the server can only send data to nodes when it is asked, therefore a request to pause or stop a node only reaches nodes that are actively asking for information. Our protocol uses regular heartbeats that may signal to stop a scan in order to overcome this problem.

Rate limiting

In some cases it is desirable to throttle the scan speed and implement a rate limit. Scanning nodes are running in user space using operating system facilities to open network connections, therefore it is difficult to apply bandwidth restrictions because there is no central place to hook in and control which requests are actually made by the kernel. Real rate limiting on packet level requires access to privileged kernel functionality which is not available in the context nodes are running and may need different implementations for different platforms. Instead, it is possible to limit the number of workers starting a scan of a target per second using a token bucket limiting algorithm [13]. For TCP and UDP a scan only consists of opening a connection to a port on a target and trying to read data within the timeout window. Higher layer scans may perform multiple requests, for example in case of HTTP redirects, but this overhead is supposed to be tolerable.

Scan interruption

Important features are the possibility to stop a scan and the so called "dead man's switch", stopping a node in case the server connection is lost. Because the scan is performed by different distributed nodes there has to be a mechanism to stop them as fast as possible in case a user wants to cancel the scan. To achieve this, the response to every heartbeat that is regularly sent by any node contains a flag signaling the node if the scan should be paused or if the node should be stopped. In case the server goes down, it is not desirable that node processes stay alive and pollute networks with their connection attempts. If a node loses connection to the server, it immediately stops the scan and exits if the server connection is not reestablished within a configurable time frame.

4.2 Architecture

The prototype consists of a server and nodes, communicating via TCP with optional TLS 1.2, c.f. Figure 4.1. This section describes the architecture on a higher level whereas the implementation section, c.f. section 4.3, gives deeper insights into concrete realization and typical control and data flow.

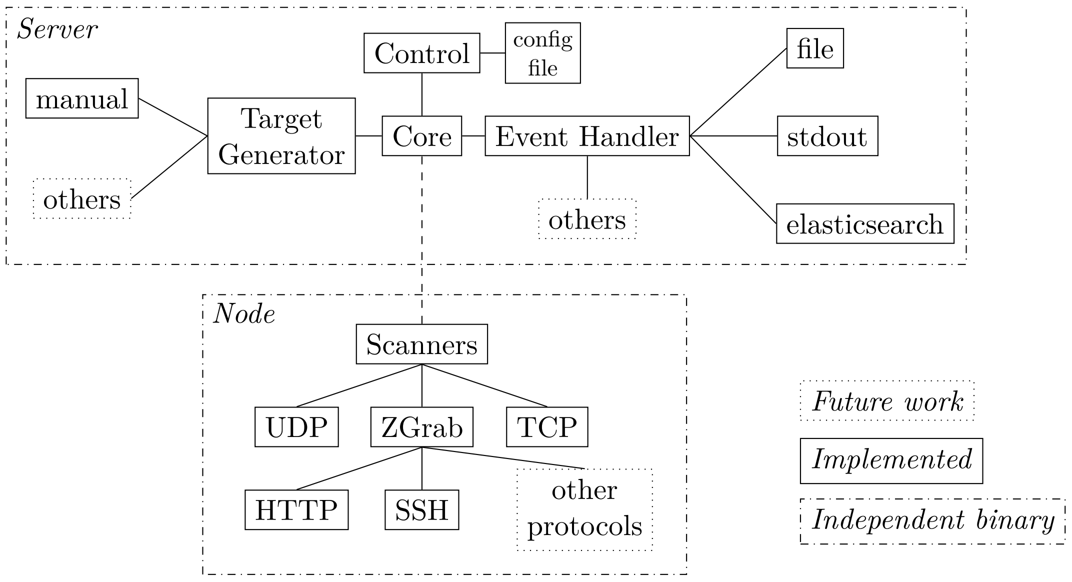

Figure 4.1: Architecture of the prototype.

4.2.1 Server design

The server is structured into several modules around a core. The core's job is to set up the program, e.g. binding to ports and initialization of resources like pools, to handle all communication and to manage nodes. Currently there is no functionality for user interaction during a running scan apart from stopping the scan, so for now the control module solely reads configuration and passes it to the core. The configuration is provided as a configuration file in the YAML format. This makes it easy for humans to read and write as well as comment it and the format has enough adaption in other projects to have plenty of parsers available. After the core is started and the configuration is parsed, the information which hosts and ports should be scanned and which targets have to be blacklisted is passed to the target generator. This module creates work batches with a list of hosts and ports to scan and sends them to the core. In case whole network ranges have to be scanned, the target generator uses the ZMap algorithm [8] for efficient on demand target generation in a pseudo-random order. Events sent back by nodes are asynchronously passed to all registered event handlers without further inspection. Event handlers implement ways of processing and storing events. It is possible to print events to standard output as well as storing them in a text file or submitting them to elasticsearch. We tried to keep the workload for adding new event handlers as low as possible[1].

[1] Implementing the elasticsearch event handler took less than an hour and is less than 200 LOC.

4.2.2 Node design

The node is supposed to be placed anywhere in the network and performs the scan.

General operation

As described earlier in section 3.1, apart from a network connection to the server and sufficient computing resources (CPU and RAM) the node must not rely on any special environmental requirements like shared libraries, elevated privileges, hard disk space or write access to certain directories. Stopping the node is not supposed to leave any file system artifacts[1]. The server address and port are passed as arguments or may even be provided at compile time in order to produce a binary that connects to a default server if no other options are supplied. After parsing the arguments, the node attempts to connect to the server and registers itself. The server sends configuration information like the number of workers to use, which scan modules to enable and which configuration to use for each module, for example timeouts. If the registration is successful, the node sends an event containing basic environment data like host name, operating system, the user running the binary, the PID of the process as well as CPU information. This is useful in order to trace a node, get insight on where nodes are running or for debugging purposes. When the environment information is sent, the node enters its main loop that requests work, spins up the scan workers, performs the scan, collects all events and reports them back.

Performing parallel port and application layer scans

Scan workers are *goroutines* reading from a channel to which work is sent as closures. These closures are functions that are completely prepared and just need to be called and may perform an arbitrary scan, so each scan module benefits from the parallelism achieved by using hundreds of workers. TCP and UDP scans use Go's built in functionality to use these protocols. Scans of higher layer protocols are done using ZGrab2 [42]. ZGrab2 is a standalone tool for performing various protocol scans, but it is written in Go and open source software which makes it possible to bundle and call the relevant scan functions directly with some glue code. Additionally, the results can be serialized to JSON in a native way, making it easy to integrate them into the event system. The prototype has a user friendly and flexible mechanism to decide when a higher layer protocol scan should be performed. The user may specify an open port condition, for example the SSH scanner should be invoked when `tcp/22`, the standard SSH port, is found to be open.

[1] Except of course the binary itself and, if used, certificates and keys required for TLS.

```
1 zgrab2:
2   enabledModules: ["ssh"]
3   # Configuration for ssh scanner
4   ssh:
5     subscribePorts: ["tcp/22"]
6     timeout: 2500ms
7     ClientID: "SSH-2.0-Go"
```

Listing 1: Snippet from configuration file

This can be configured in the configuration as shown in Listing 1, line 5, and during the initialization, protocol scanners register themselves for events involving this port. When the TCP scanner discovers an open port `tcp/22`, the SSH scanner is notified, the scan function is prepared and queued along with all other port and protocol scans. It is possible to define an arbitrary number of TCP and UDP ports for each scanner to be notified on and there are possibilities to further improve this mechanism, for example by notifying based on grabbed banners.

4.3 Implementation

This section describes the concrete implementation details of the important parts as well as relevant and probably interesting technical details.

4.3.1 Control and Configuration

The prototype relies on well known libraries for argument and configuration parsing, `cobra` [31] and `viper` [32]. Both are developed by the same author and are adopted by large projects like Kubernetes, Hugo, rkt, etcd or Docker. Cobra is used for command line parsing and allows to build flexible and powerful interfaces. Apart from its main task, it allows to automatically generate man pages, help text, suggestions in case of typos or shell auto completions. Usage of command line flags is minimized and they are only used to control the initiation of server and node components, e.g. where to find configuration files in case of the server or to provide server location and TLS artifacts in case of nodes. Nodes are supposed to operate standalone, therefore all configuration is passed via command line flags or sent by the server.

Viper is used for server side configuration file parsing. Viper supports various configuration file formats (e.g. JSON, TOML, YAML and others) as well as obtaining configuration from remote key value stores, live monitoring of configuration changes, configuration from environment variables and it integrates seamlessly into Cobra. In our use case it comes in handy because it supports extraction of configuration sub trees

```yaml
# ...
scannerconfig: # This configuration node is extracted and sent to nodes
  # This is global node configuration
  workers: 1000
  ratelimit: "none"

  # This configuration node is again extracted and passed to the TCP scanner
  tcp:
    # ...
    timeout: 2500ms

  # The UDP scanner may use different settings
  udp:
    # ...
    timeout: 800ms

  # The options for higher layer scans are defined here
  zgrab2:
    # Depending on which modules are enabled, configuration sub trees
    # are dynamically extracted and passed to the respective scanners.
    enabledModules: ["ssh", "http"]
    ssh:
      subscribePorts: ["tcp/22"]
      timeout: 2500ms
      ClientID: "SSH-2.0-Go"
      # ...
    http:
      subscribeHTTPPorts: ["tcp/80", "tcp/8080", "tcp/8000"]
      subscribeHTTPSPorts: ["tcp/443", "tcp/8443"]
      timeout: 2500ms
      # ...
# ...
```

Listing 2: Hierarchical configuration

and (de)serialization of configuration data. This is used for example to extract scanner configuration that is sent to all nodes, c.f. Listing 2 for an example.

There is configuration affecting the basic operation of the node, like number of workers or rate limits, and there are options allowing fine-grained configuration of each scanning module, e.g. which timeouts to use for a scan. Each scanner only gets its own configuration sub tree, enabling scanner specific configuration while reusing code by constantly referring to the same configuration options like `timeout`.

Sometimes it is desired to have a default configuration baked into the binary. Go allows setting uninitialized string variables during linking[1] and the node has explicit support of this mechanism for setting a default remote server and port to connect to if no other options are provided.

4.3.2 Core

The core implements the central functionality for node management and communication. It connects and controls all other modules and holds a global data structure where configuration information like ports to listen on and TLS configuration as well as internal information like pooling and event handlers are stored.

Message Queue

The prototype relies completely on the `mangos v2` framework [21], a brokerless message queue implemented in pure Go. Mangos brings support for several transports like TCP, web sockets or IPC with optional TLS if desired. In contrast to regular TCP, mangos implements a set of schemes called *scalability protocols* that model classic communication schemes, like bus, publisher-subscriber, pipeline, surveys across nodes, or conventional request-reply patterns. The main reason behind choosing mangos over plain TCP was message based communication. Other benefits are features like easy integration of message security (TLS, maybe other protocols like SSH in the future), easier and more flexible handling of connection loss and reconnections as well as the fact that it has already proven to be reliable because it is used successfully by other projects. Another small but maybe helpful feature is that mangos allows binding to multiple ports, so it is possible to simultaneously listen on e.g. ports 22, 80, 443 and others. Depending on the access restrictions of the networks nodes are placed in, each may choose any of these ports to reach the server, allowing easier firewall traversal. The prototype uses the `REQ REP` scalability protocol as well as the `tcp` or `tcp+tls` transport.

Pooling

The server core implements pooling, which is the central mechanism behind the work distribution logic. Each node registering at the server is placed in a pool and each pool is performing a complete scan of all targets. This means that placing multiple nodes inside the same pool splits the work across nodes and speeds up the scan whereas multiple pools can be used to scan targets multiple times. For example, hosts scanning

[1] via the linker's -X flag [35], for example defining a default server and port looks like this: `go build -ldflags "-X main.server=10.0.0.1 -X main.port=8601"`

from an internal network and hosts scanning from an external network are placed in different pools and the scan results contain information about network visibility from different vantage points. Each pool has its own target generation, therefore different pools scan targets in a different order and pools with different scan speed do not affect operation of other pools. Currently, placing nodes in pools happens by fair distribution, meaning a node is placed in the pool with the fewest members, or by passing an explicit pool preference parameter when invoking a node.

Job handling

Targets generated by the target generator are packed into work batches called jobs, which are also managed by the pool. Since jobs must not be assigned multiple times whereas each job must be worked on, there is a job area where jobs are stored. If a node asks for more work, a job that has not been assigned is taken from the area, the name of the node working on the job as well as the time when the job was assigned are stored and a timeout is defined. If the timeout is reached and the node was too slow or has been lost, the metadata is cleared from the job which implies that it is going to be reassigned. In case the node reports results for a job, the server checks if the node is still supposed to work on the job and if this is the case, the events are sent to all event handlers, the job is deleted and the node receives an acknowledgement for its honorable work.

Message format

Messages are defined as Protocol Buffers, a data serialization mechanism developed by Google [27]. Messages are defined in a schema and a compiler creates Go code for serialization and deserialization of messages. This is an easy solution that also provides smaller messages than text based encodings like JSON. Listings 10, 11 and 12 show all defined messages and their purpose.

Stopping nodes

The `REQ REP` scalability protocol does only support sending messages as reply to a previous request. Therefore nodes are sending regular heartbeats to signal a server that they are still alive and in case a node needs to be paused or stopped, the server has the chance to set special flags signaling to pause or stop in the response. A node that is inactive for a certain amount of time is removed from the according pool and needs to register again. When target generation is done and all jobs are finished or when the user hits `Ctrl+C`, the server waits for all nodes to send a heartbeat, signals them to stop and once every node has stopped all event handlers are closed and the server is shut down.

4.3.3 Target Generation

The target generator determines which hosts to scan based on the target specification of the user. In the simplest cases, this is either a domain name or an IPv4 address, the third option is to specify a CIDR range of a network. There is no restriction on the number of targets and mixing networks, IP addresses and domain names is possible, too. The target generator decides based on a regular expression if the input is to be treated as a network or IP address. Unfortunately it is difficult to decide if a domain name is valid, therefore the target generator only does a simple check for absence of colons and slashes because they have a special meaning in a FQDN context and should not be present. If no colons or slashes are found, the target generator is tolerant and assumes the name to be valid. If CIDR notated networks are passed as targets, the target generator uses a native Go implementation of the algorithm used by ZMap [8]. This algorithm simply generates a pseudo random generator of a cyclic group and the generated elements are mapped to the available IP addresses in the network. This way all IPs are addressed exactly once without a particular order or the requirement to keep all IPs in memory.

The target generator also has to deal with port definitions. This may be single ports, port ranges like $1 - 1024$ or common top ranges. Nmap curates a list of top ports, sorted by their frequency to be found open according to the Nmap project [22]. We also integrated this list of TCP ports and they may be used by specifying *topN*, e.g. *top50* as ports to scan. The target generator takes care of removing duplicates, therefore the port specification `top50, 1-1024, 80` would only contain port 80 once although it is also element of the top50 and $1 - 1024$ sets.

Blacklisting is also performed during target generation. Similarly to target specifications, the user may supply any IPs, networks or domain names for blacklisting. Networks are stored in an IP-based prefix tree using a native Go implementation of the ZMap authors and single IP addresses are encoded as /32 networks [43]. Blacklisted domain names are stored in a map because later on each access is querying for key existence which is a fast operation.

4.3.4 Node and Scanners

Like the server, nodes use Cobra for argument parsing. Nodes are supposed to be self contained and work autonomously, therefore there is no configuration file for nodes. Options like server address and port are passed as arguments while scan settings are specified in the server's configuration and obtained from the server. After parsing arguments, the node tries to connect and register at the specified server. The registration message contains a unique machine ID that may be used to force a single node to run on a machine if desired. Apart from the machine ID, a node name may be specified as well as a preferred pool and information about the environment. In response to the registration message, a node ID is assigned by the server and configuration for scanners is provided. The timestamp sent by the server is used to calculate an internal clock offset that is added to each event, this way clocks that are wrong are compensated.

The configuration obtained from the server is used to initialize an internal data structure holding all information required for the scan: node name, ID, the time offset, scanner configuration as well as all channels used to pass messages, the rate limiter and the list of scanner subscriptions. The scan subscriptions, the mechanism used to realize on demand higher layer scans as described in 4.2.2, have the interesting type shown in Listing 3 which is a map that maps a string to an array of functions that expect a protocol string, a host string, a port number and a channel to send events to while these functions must again return functions. This may look like an unnecessary complex type, but allows to simplify scanning logic later on. Assume the HTTP scanner is supposed to be invoked every time an open port `tcp/80`, `tcp/443`, `tcp/8000` or `tcp/8080` is discovered.

```go
map[string][]func(proto string, host string, port uint, result chan<- *Event) func()
```

Listing 3: Type holding scan subscriptions.

The map is going to contain these strings as keys and their values contain functions that are used to prepare the closures that are actually performing the scan later on. During a scan, the functions stored in the map are called with the information where the open port was discovered, protocol, host and port number, as well as where further scan results may be reported. Instead of performing the scan, a new function closure containing all information is returned which only needs to be called by a worker to perform the scan. This way it becomes possible to define arbitrary scan functions that are triggered only if certain open ports are found and these scan functions still benefit from the worker architecture and performance gains by running concurrently because workers do not have to know about concrete scanning logic.

Concurrent scanning

After registration, nodes immediately start with requesting work. If no work is available, the scanner waits a few second before requesting new work. After receiving work, the code shown in Listing 4 spins up the desired number of workers. The ability to spawn a large number of workers as asynchronous goroutines is the sole reason for the scan performance of each node. The wait group `wg` is Go's native mechanism to keep track

```
1  // Spin up workers
2  // Each worker has access to ScanController's work queue
3  // Work queue contains functions that are fully prepared,
4  // this means they have full state regarding targets,
5  // timeouts, configuration, where and how to report.
6  // Workers are just here to control the level of concurrency
7  var wg sync.WaitGroup
8  for i := 0; i < controller.scannerConfig.GetInt("workers"); i++ {
9    wg.Add(1)
10   go func(queue <-chan func()) {
11     for queuedTask := range queue {
12       atomic.AddInt64(&controller.scansRunning, 1)
13       controller.ratelimiter.Wait(context.TODO())
14       queuedTask()
15       atomic.AddInt64(&controller.scansRunning, -1)
16     }
17     wg.Done()
18   }(controller.scanQueue)
19 }
```

Listing 4: Most code responsible for spinning up workers and perform scans.

of how many related goroutines are running and makes it possible to wait until all goroutines signaled that they are done by calling `wg.Done()`. Unfortunately, wait groups are only meant to block execution until all workers are done and do not allow to obtain the exact number of currently running workers, therefore line 12 and line 15 contain additional code to keep track of running scanners which is relevant for tearing down later on. Workers are anonymous functions that are spawned asynchronously in line 10. Their only argument is a channel where they are able to read more work in form of closure functions (line 11) that do not expect parameters and do not return any value. These functions may wrap arbitrary scanning functionality, for example simple TCP or UDP scans or more complex scans like HTTP with TLS enabled. These functions have been prepared beforehand and have all required parameters like target host, port and how to report results already included, therefore there is no need to deal with it here which massively simplifies the code at this point. If the rate limiter's `Wait()` (line 13) has been passed, the worker is able to perform the scan just by calling the function. Workers repeat this until the work channel is closed, then the work loop is stopped and the worker functions return.

TCP scanning

TCP port discovery uses Go's standard networking interface to open a TCP connection to each port. If the connection is successful, the port is marked open and if the connection is refused or times out, the port is marked as closed. Since standard operating system interfaces are used, a full TCP hand shake is performed and each connection is teared down cleanly instead of resetting the connection once the peer answered, the standard behavior of Nmap's SYN scan.

UDP scanning

Scanning UDP is a bit different to TCP. Because UDP is a stateless protocol, closed ports usually return an ICMP message telling that the service is unreachable, but these may be filtered or rate limited by a firewall. They also only tell that a port is closed, therefore absence of these messages is only an indicator for an open port, but the only way to be sure is to trigger an actual reply from the requested port. Replies to packets may be protocol dependent and replies may only be sent to valid packets, therefore the prototype includes an UDP protocol payload list taken from the Metasploit framework [19]. For example, if the DNS port 53 is scanned via UDP scan, a valid DNS packet is sent in order to increase the chance to receive a reply. It is possible to configure

```
udp:
  # Fast sends only probes for known protocols
  fast: false
  # Default payload. This is sent when the scanner is not aware of the protocol.
  defaultHexPayload: "\x41\x41\x41" # 'AAA'
  # You may define/overwrite port:payloads at your wish.
  # For encoding arbitrary data, see https://golang.org/ref/spec#Rune_literals
  customHexPayloads:
    "19": "A" # chargen. "A" is the same as "\x41" (hex) or "\101" (oct)
  # Timeout to wait for a response
  timeout: 800ms
```

Listing 5: UDP payload configuration.

the default payload to send in case there is no default payload available as well as custom payloads for specific ports using the configuration file. Listing 5 shows an example of the configuration that sets the default UDP payload to `AAA` using hex notation of the payload and UDP 19, the chargen protocol, to a single `A` character which is a valid payload for that protocol [25]. That mechanism allows to provide payloads for UDP services in a simple way without recompiling and redeploying all nodes.

Higher layer scanning

As already explained, the prototype has the possibility to perform subsequent higher layer protocol scans in case certain ports are found to be open. Currently, the ZGrab2 [42] framework, supporting over 20 protocols like IMAP, FTP, HTTP, SSH, SMB or several database protocols, is used. Because ZGrab2 is a command line application instead of a library, some glue code is required to integrate a ZGrab2 protocol scan into our way of scanner configuration, execution and result reporting. Using a hooking mechanism, it is possible to configure when application layer scanners should be invoked. Currently SSH and HTTP(S) scans are implemented, other protocols will be integrated over time.

4.3.5 Event Handler

Each time a scanner module discovers something an event containing the finding is generated. These events contain all relevant information, for example where and when it was generated by which scanner, which host was scanned and all scan result data. The event based approach allows to work with scan results even if a scan is still ongoing and enables insight into changes over time. Because usage scenarios, relevance of data as well as the amount of data generated may differ significantly, we tried to create a generic interface to process event data.

```go
type EventHandler interface {
  Configure(*viper.Viper) error
  ProcessEvents([]*schema.Event)
  ProcessEventStream(<-chan *schema.Event)
  Close() error
}
```

Listing 6: Event Handler Interface

On top of this interface, all available output modules are built. This currently includes printing to standard output, writing to a file and sending data to an elasticsearch endpoint. Implementing this interface is easy and straightforward, therefore other flavors like XML export, sending data to a SQL server or any other big data backend is supposed to be realizable within hours. Each event handler implementation may require its own configuration. This is done in the configuration file and the respective configuration sub tree is passed to each interface. This gives the event handler a chance to set up the environment, for example preparing files or connections to remote endpoints or databases. During a scan, events may be sent using either `ProcessEvents()` or `ProcessEventStream()` functions. The existing implementations of these functions spawn asynchronous tasks that process the results, this way the function returns early and nodes do not have to wait for data to be processed by a backend. `Close()` is called when the server is going to stop because the scan is done. It is supposed to take care that all data is processed and all file descriptors or connections to other backends are closed. The function is supposed to block until data is persisted and the server is allowed to stop.

The EventHandler interface supports very simple event filtering capabilities that may be configured separately for each event handler. Listing 7 shows the complete implementation of the filter functionality. This is especially useful for filtering terminal output,

```go
// FilterMatchesEvent returns true if the event has the filter
// string and its value matches the provided value
// Configuration may look like this:
// portscan.port: 1099 // shows port scanner events with port 1099
func FilterMatchesEvent(event string, filter string, value interface{}) bool {
  if value == nil {
    return gojsonq.New().JSONString(event).Find(filter) != nil
  }
  return gojsonq.New().JSONString(event).Find(filter) == value
}
```

Listing 7: Implementation of simple event filtering.

for example to mute all events except those that are of increased interest and meet a certain condition, e.g. Java RMI services usually listening on TCP port 1099.

Chapter 5

Evaluation

This chapter evaluates the capabilities of our prototype and compares it to other, similar tools.

5.1 Testbed evaluation

When creating software from scratch, it is important to make sure it works as expected and is functional. To evaluate this, a testbed with two networks, a router, some machines and active firewall rules was created. This also allows to introduce real world conditions like limited bandwidth, delay or packet loss. Because the number of hosts and open ports is well known and can be controlled, scans can be performed and reproduced at any time. The following subsections describe the testbed setup and results by performing scans with limited bandwidth and variations in packet loss.

5.1.1 Setup

The testbed uses mininet[1] on the official virtual appliance running Ubuntu 14.04 provided by the project. Due to a bug in the TCLink (traffic controlled link) implementation [20], pull request #863[2] was applied by hand. The fix for the bug has been merged upstream but the virtual appliance was not updated since, hence the manual patch was required. Listing 8 shows the script for setting up the mininet.

[1] `http://mininet.org/`

[2] `https://github.com/mininet/mininet/pull/863`

In the hypothetical scenario, the network consists of 198 hosts, 99 in an office network and 99 in a remote data center, connected via a Linux router. The remote connection simulates an Internet link with 25 Mbit/s which is about the mean Internet link speed in Germany [41]. Figure 5.1 shows a simplified schema of the testbed. The office network consists hosts exposing TCP ports typical for Microsoft Windows machines: 445 (SMB) and 139 (NetBIOS). Additionally some of these machines are mimicking development or test machines, having open ports like 3306 (MySQL), 3389 (Remote Desktop) or 8000 and 8080 that are often used as alternative HTTP ports. Some also imitate Domain Controllers and open ports 88 (Kerberos), 389 (LDAP), 445 (SMB) and 636 (LDAPS). Hosts running in the data center location are supposed to look like a mixed Linux and Windows server environment, exposing an additional set of TCP ports: 22 (SSH), 25 (SMTP), 80 (HTTP), 110 (POP 3), 143 (IMAP), 443 (HTTPS) 465 (SMTP over SSL), 587 (SMTP, usually encrypted), 993 (IMAPS), 995 (POP3S) and FTP (20 and 21). The router between the networks is configured to allow any connection to pass from office to the data center network as well as it permits already established connections to pass back. Connections originating from the data center network are restricted to access only a limited set of hosts and ports in the office network. Listing 9 shows the rule set configured on the router. The services were simulated by listening ncat[1] instances that do not do anything except accepting the incoming TCP connection, therefore higher layer protocol scans are not possible in this testbed setup.

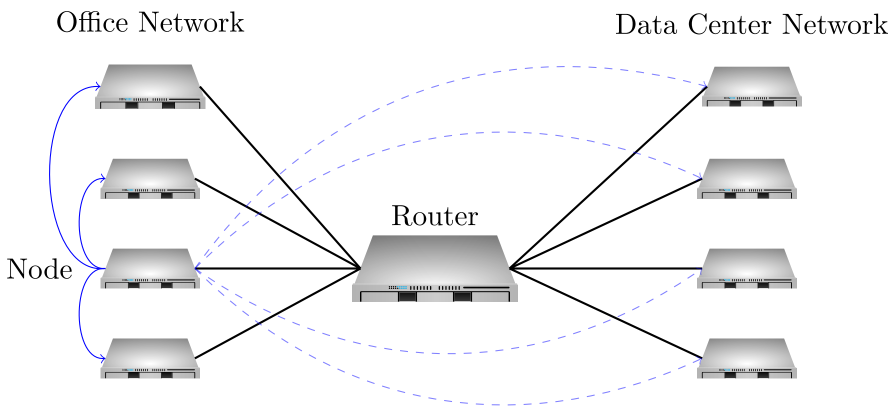

Figure 5.1: Simplified schema of the testbed with 4 hosts in each network and a node placed in the office network performing a scan.

[1] `https://nmap.org/ncat/`

We scanned both networks from vantage points in both networks with different packet loss values (0.001%, 0.01%, 0.1%, 1%, 2%, 5%, 10%, 20%, 50%). Cubic TCP congestion control algorithm already shows throughput difficulties with packet loss >0.01% and Google's BBR algorithm begins to have problems from 10% upwards [3], therefore it is not expected to have such high packet loss in operational real word networks.

In total, the office network in the testbed consists of 80 discoverable ports as well as 89 discoverable ports in the data center network. The port scan is supposed to find all ports in both networks when scanning from the office network whereas the scan originating from the data center should be able to see all open ports in the data center, but only 4 explicitly allowed ports in the office network. Because the link between office and data center is the only link with limited bandwidth and packet loss enabled, each scan run should find all open ports in the same segment where the scan is originating.

5.1.2 Results

In all scans, one for each vantage point, the prototype was able to find all ports in the respective local network. The scans originating from the data center network that were supposed to find the 4 open ports in the office network were successful for all packet loss configurations except 20% and 50%, finding only 3 (20%) and 1 (50%) open ports respectively. Figure 5.2 shows the results of the scan originating from office network to data center network. In this setup, all discoverable ports were found with packet loss up to 5% and 10% packet loss still allowed to find 88 of 89 possible targets. These testbed results show that our prototype is suited to perform a port scan with accurate results even under suboptimal environment conditions like limited bandwidth and packet loss.

5.2 Real world evaluations

In order to evaluate the prototype against larger networks under more realistic conditions, some measurements under real world conditions were performed. We performed three measurements, M1, M2 and M3 against the network of the Rechnerbetriebsgruppe (RBG)[1] of the faculties for mathematics and informatics at TUM. They allowed to scan their /16 network from the Internet, whitelisted the IPs of the servers used for scanning and provided a list of networks to exclude, for example the IP address pool used for assigning IPs in the eduroam network. In total, exclusions taken into account, 53248 hosts were in scope for scanning. They also supplied a solution to trigger the

[1] `https://www.rbg.tum.de/`

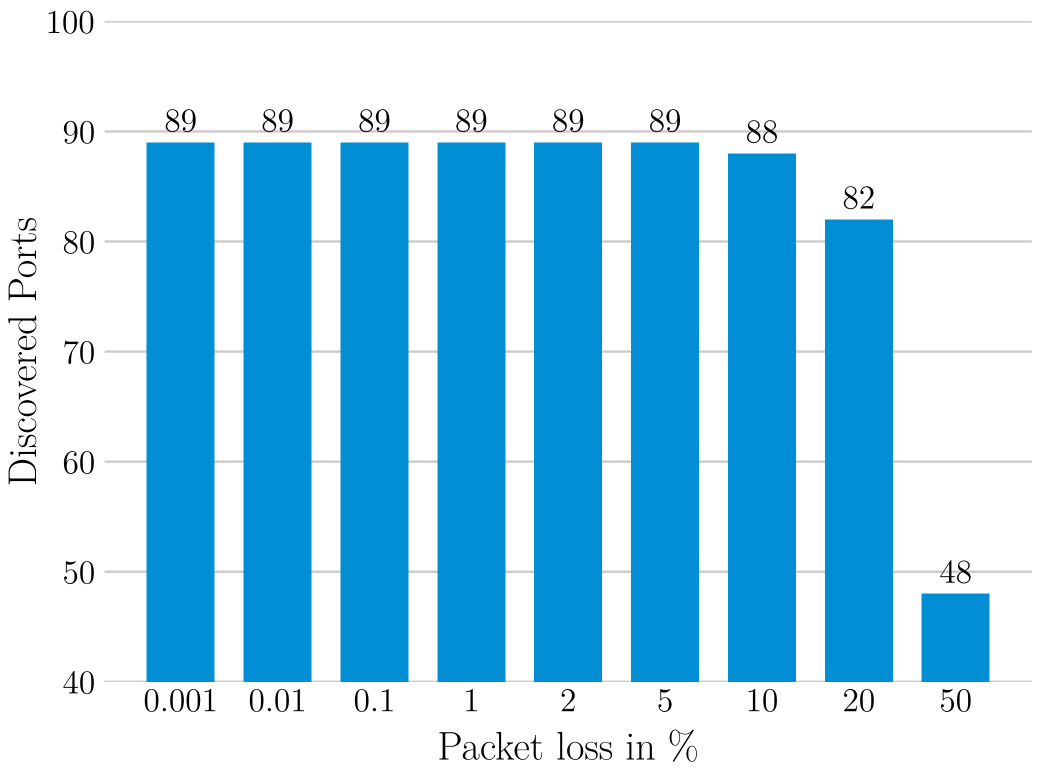

Figure 5.2: Discovered ports when scanning from office to data center network with various packet loss configurations.

scanner binary at any time to perform the scan from within their network. Scans from the Internet were conducted using several virtual cloud servers hosted by a large German hosting company. They are equipped with two vCPUs running at 2.1 GHz and four GB RAM each and the instance cost is about €6 per month. According to the hoster's documentation, all cloud servers are equipped with a gigabit link. All servers run Debian 9[1] with the latest patches to the date of the first measurement applied. No firewall rules were in place on the virtual machines and the hoster confirmed that there is no central firewall installed. All tools and scanners used (Nmap, ZMap and Masscan) were installed from Debian's repositories, apart from our prototype no software was (re)compiled or customized by ourselves.

[1] `Linux 4.9.0-8-amd64 ##1 SMP Debian 4.9.144-3.1 (2019-02-19) x86_64 GNU/Linux`

The measurements M1, M2 and M3 were performed in the night hours on weekends in order to avoid irregularities caused for example by daily business operation. The port list scanned was the top 50 port list provided by Nmap[1].

5.2.1 Scan from external and internal vantage points

In order to compare the effectiveness of scans from different vantage points, the prototype was deployed on a virtual server on the Internet and running as restricted user on a workstation of a RBG member. The same network range was scanned from both positions, one at a time.

General results

Table 5.1 shows the results of both scans. The measurements allow detailed inspection of found ports and services. A full analysis was omitted from this thesis for scope and time reasons, but some interesting findings are described further in this section.

One can see that administrative interfaces like SSH (22) and RDP (3389) are about three times as common from inside the network as they are visible from the Internet. Network ports like SMB (445), Windows RPC (135) or NetBIOS (139) are usually found on Windows machines and should probably not be open to the Internet. For example Eternalblue, the exploit stolen from the NSA and used in the WannaCry ransomware campaign, used an unauthenticated remote code execution vulnerability in SMB and was therefore able to infect millions of systems that were connected to the Internet [23]. The measurements show that almost none of these services are found from the outside, but are widespread from the inside. This is probably due to firewall rules enforced at a central network position preventing external access to these ports. Similarly, services like MySQL (3306), VNC (5900), Telnet (23), unencrypted SMTP (25) or DNS (53) may impose security risks and should only be reachable via the Internet on purpose.

Figure 5.3 shows the distribution of ports discovered in the first measurement using our scanner. It clearly shows that the majority of services are SSH (22) and HTTP(S) (80 and 443) with other HTTP ports (8000, 8080) and remote control (RDP, 3389) trailing behind.

Because the discrepancies are not that high compared to protocols like SMB, it is likely that these instances are on purpose Internet connected and are specifically whitelisted

[1] 80, 23, 443, 21, 22, 25, 3389, 110, 445, 139, 143, 53, 135, 3306, 8080, 1723, 111, 995, 993, 5900, 1025, 587, 8888, 199, 1720, 465, 548, 113, 81, 6001, 10000, 514, 5060, 179, 1026, 2000, 8443, 8000, 32768, 554, 26, 1433, 49152, 2001, 515, 8008, 49154, 1027, 5666, 646

	M1		M2		M3	
	int	ext	int	ext	int	ext
21	19	12	19	12	18	11
22	1846	523	1821	531	1825	524
23	8	1	7	1	8	0
25	105	5	103	4	100	4
26	7	0	7	0	7	0
53	29	8	30	8	30	8
80	640	509	651	514	645	515
81	4	3	3	2	3	2
110	10	6	11	7	11	7
111	308	42	307	42	311	45
113	3	2	3	2	3	2
135	83	1	213	1	211	0
139	94	0	227	0	228	0
143	17	13	18	14	18	14
179	2	2	2	2	2	2
443	502	394	506	400	506	401
445	101	0	237	1	232	0
465	26	25	25	24	25	24
514	2	0	2	0	2	0
515	38	1	38	2	38	2
548	5	1	6	2	6	2
554	2	0	2	0	3	0
587	23	23	23	22	22	22
646	2	2	2	2	2	2
993	21	17	22	18	22	18
995	13	9	14	10	14	10
1433	2	1	2	1	1	0
1720	1	0	1	0	2	2
1723	5	5	5	5	4	4
2000	10	6	10	6	10	6
3306	28	12	26	12	25	11
3389	146	53	156	51	148	49
5060	3	2	3	2	2	1
5900	17	2	17	2	19	2
6001	2	1	2	1	2	1
8000	94	93	91	90	90	89
8080	41	31	38	27	39	27
8443	3	2	3	2	3	2
8888	3	2	2	1	2	1
10000	1	1	1	1	2	1
49152	2	1	2	1	2	1
49154	10	4	10	4	10	4
Total	4278	1815	4668	1827	4653	1816

Table 5.1: Ports discovered by scanning from internal and external (the Internet) vantage points.

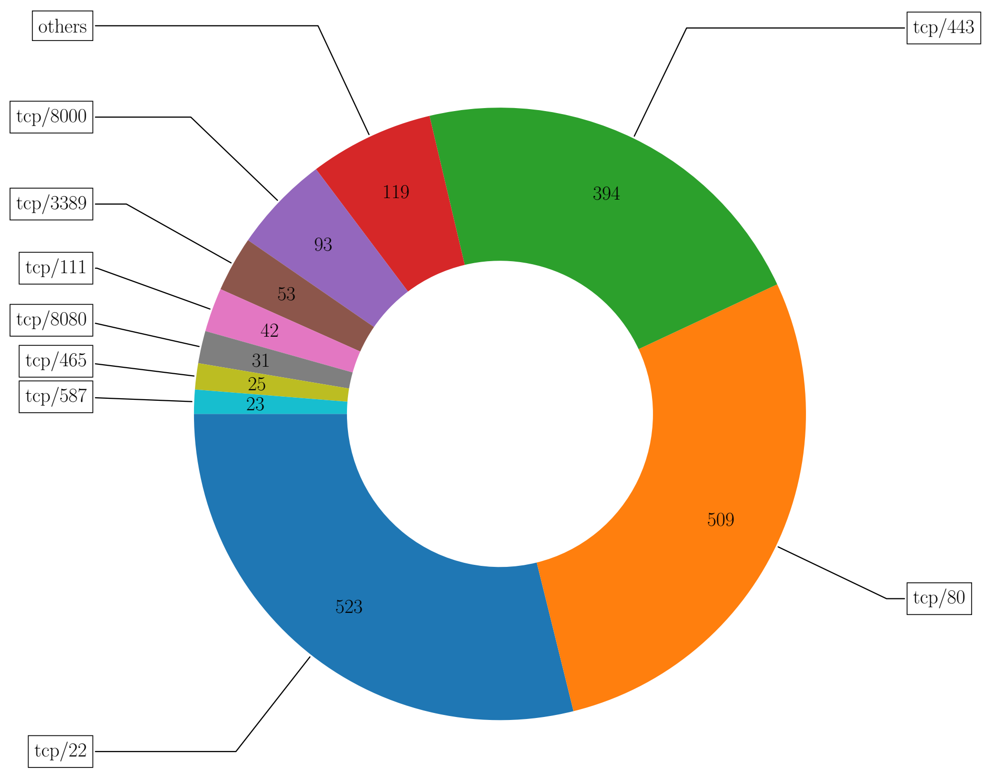

Figure 5.3: Ports discovered by scanning from the Internet using our prototype, measurement M1.

if a central firewall is in place for these protocols. SSH and RDP are common remote administration protocols and the RBG network is a heterogeneous environment with lots of stakeholders. Concluding from the high number of Internet facing SSH and RDP ports, it is likely that there is no or only partial central filtering for these services and many of them are unintentionally reachable from the Internet. The same goes for services like HTTP (80) and HTTPS (443) as well as alternative ports usually running these protocols like 8000, 8080, 81, 8443, 8888. They are, if at all, only partially filtered centrally.

Analysis of TLS certificate data

Because the prototype has integration of some ZGrab2 modules it is possible to compare data on higher protocol levels, for example TLS. Evaluation of TLS certificate data is a typical use case we are going to focus on in this section, nevertheless the prototype can be extended to analyze more protocols, for example SSH, SMB, BACnet and others. HTTP scans during the measurements were configured to follow HTTP redirects and retry HTTPS connections if HTTP connections fail. This leads to some vagueness of the following results because the protocol scan also follows redirects to other domains, e.g. Github pages, or scans may be contained twice, first for HTTP on port 80 that may have redirected to HTTPS on port 443 and another independent scan directly targeting port 443 with HTTPS. Also, responses that exceeded some other restrictions like unexpectedly large responses were not further analyzed and therefore the results presented regarding TLS only scratch the surface. Nevertheless, the results may give an idea of the big picture as well as on scan capabilities and data evaluation possibilities provided by the prototype. Using `jq`[1] and several queries (see Listings 13, 14, 15, 16) it is easy to gain a fast insight on various interesting metrics.

First, we were interested in certificates issued by CA's that are unusual for usage at TUM. It is possible to obtain certificates from the *Verein zur Förderung eines Deutschen Forschungsnetzes* (DFN), an organization operating research networks in Germany [6]. These are trusted by all browsers and there are processes that are supposed to make sure that these certificates are only issued to people responsible for systems, e.g. by checking their ID. Additionally, TUM and LRZ, the data center providing most IT services for universities and many other research institutions in Munich and Bavaria, have CAs that are assumed to be legit. Finally, we accept Let's Encrypt certificates, issued by the free Let's Encrypt CA that is widely used across the Internet and that is also commonly seen at servers operated at TUM. Since these CAs are trusted by browsers and certificates can be obtained at no cost, we are even interested in other trusted CAs used because there may be people not knowing about DFN and Let's Encrypt that instead buy certificates from other CAs, potentially with tax money.

	M1		M2		M3	
	int	ext	int	ext	int	ext
Unusual CAs	117	64	120	65	122	62
DFN / TUM / LRZ	279	236	274	236	280	242
Let's Encrypt	65	68	70	71	71	73
Total	461	368	464	372	473	377

Table 5.2: Certificates issued grouped by CAs.

[1] `https://stedolan.github.io/jq/`

	M1		M2		M3	
	int	ext	int	ext	int	ext
2006	1	0	1	0	1	0
2007	2	1	2	1	2	1
2008	0	0	0	0	0	0
2009	2	2	2	2	2	2
2010	1	1	1	1	1	1
2011	1	1	1	1	1	1
2012	2	1	1	0	2	1
2013	2	1	2	1	2	1
2014	3	2	3	2	3	2
2015	1	1	1	1	1	1
2016	19	4	18	4	18	3
2017	13	6	12	7	13	7
2018	14	11	12	10	12	10
2019	3	3	6	6	4	4
Total	64	34	62	36	62	34

Table 5.3: Expired certificates found. For each year, the number of certificates expired in this year is shown. For 2019, a certificate is marked as expired if the expiry date was before the scan date.

Table 5.2 shows the numbers of certificates discovered that were not issued by any of the CAs mentioned above. Many of those certificates are default certificates, for example default printer certificates but also Network Attached Storage (NAS) boxes or default certificates of software like VMWare. Some of the certificates seem to be issued by local CAs of chairs or individual people. Apart from wildcard certificates issued by DigiCert that belong to a company founded by a chair, it seems that no bought certificates are in use.

In total, the number of unique certificates presented by web servers, regardless of being valid or trusted, was between 461 and 473 from inside the network and 368 and 377 from the Internet.

Another interesting information stored in certificates are their life times. For example, certificates that were issued a long time ago may reveal a systems age and expired certificates are often indicators for unused, forgotten or legacy systems. It is also uncommon that browser trusted leaf certificates used for HTTPS on web servers have a long life time, e.g. more than 4 years.

Table 5.3 shows the numbers of expired certificates observed. The oldest certificate seen internally expired in 2006, followed by two certificates that expired in 2007. One of these certificates can also be observed by scanning from the Internet. These systems

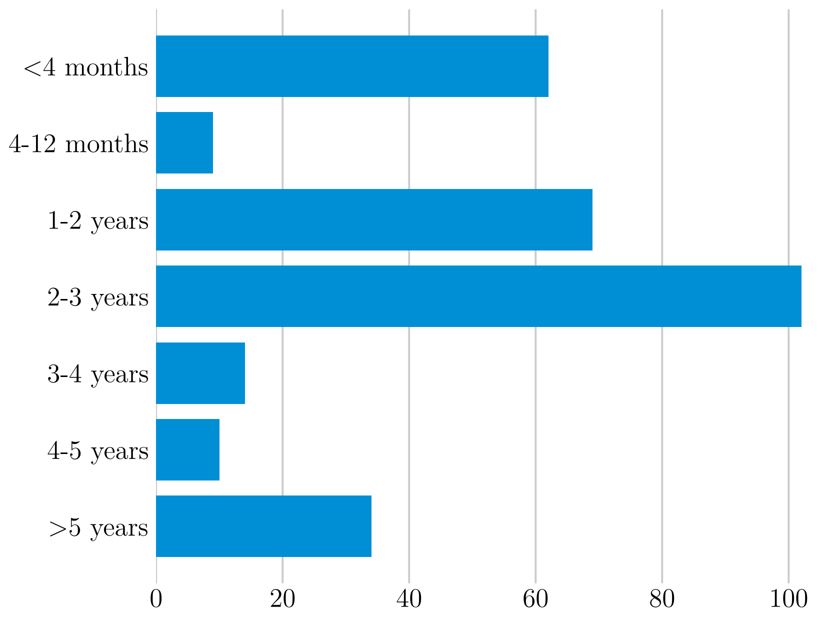

Figure 5.4: Lifetime of certificates observed in M1 from external.

may be shadow IT or completely forgotten systems that are not used anymore. The number of expired certificates is low until 2016 several internal certificates started to expire. We analyzed the lifetime of certificates observed in M1.

Figure 5.4 shows lifetime ranges of certificates and how many certificates have been observed with these lifetimes. The high number of certificates having a lifetime less than four months is due to free Let's Encrypt certificates that have a 90 day lifetime and are automatically renewed if the services are set up correctly. The majority of certificates have a life time between one and three years which is usual for manually enrolled certificates. Watching these numbers in the future may be an interesting indicator for prevalence of automatic certificate enrollment since this allows to have certificates with a shorter lifetime. Certificates that have a life time longer than five years are often default certificates that have not been changed.

The last thing in our evaluation of TLS certificates was to look for certificates that have a long list of Subject Alternative Names (SAN). These names are FQDNs the certificate is valid for and a long list means that probably many names point to a single server and many applications are running on this machine. This is usually a good way to gain insight on available systems, used names and where to dig deeper when looking for vulnerabilities.

The certificate with most SANs had a total number of 627 domain names it is valid for and seems to be used for SAP. The second highest amount of SANs were found in a certificate used by a web server that provides central TYPO3 hosting for all TUM institutions that do not want to take care of their own hosting. During the measurements 366, 370 and 373 SANs were found in the respective certificates. It seems that a new certificate with a two year validity is generated each time subdomains are added or deleted. The oldest of these observed certificates has already been revoked whereas the others were still valid but not used anymore due to high turnover. At this point there may be potential for optimization in order to improve the revocation process and keep the number of valid certificates for a given domain name low. Apart from those certificates with hundreds of SANs, some chairs are using a single certificate for all their web sites and aliases[1] which goes up to 54 for one chair.

5.2.2 Comparison to other scanners

	Nmap	ZMap	Masscan	Prototype
Scanner architecture	Stateful	Stateless	Stateless	Stateful
Operating system	All major[2]	Unixoide[3]	All major [14]	All major [12]
CPU Architecture	All major[4]	All major[4]	All major[4]	All major [12]
Programming language	C	C	C	Go
Runtime Dependencies	✓	✓	✓	✗
User privileges suffice	✓	✗	✗	✓
Result format	Text, 13375p34k, Grepable, XML	Text	Binary, Grepable, JSON, XML, List	JSON
TCP / UDP scan	✓/ ✓	✓/ ✓	✓/ ✓	✓/ ✓
Application layer scans	✓	✗	✓	✓
Builtin scan distribution	✗	✗	✗	✓
Configuration	Flags	Flags	File, Flags	File, Flags[5]
Blacklisting	✓	✓	✓	✓
Rate limiting	✓	✓	✓	✓

Table 5.4: Feature comparison between well-known port scanners and our prototype.

[1] e.g. mappings like `chairname.in.tum.de`, `www<chairnumber>.in.tum.de` in addition to those names for `cs.tum.edu` and `informatik.tu-muenchen.de` domain names, too.

[2] Windows, Linux, MacOS, BSD, Solaris, AIX, AmigaOS, various proprietary Unix distributions, see `https://nmap.org/book/install.html`

[3] Linux, MacOS, BSD, see `https://github.com/zmap/zmap`

[4] According to the availability of respective Debian packages: amd64, arm64, armel, armhf, i386, mips, mips64el, mipsel, ppc64el, s390x

[5] Flags allow only limited configuration, for example where to find the server

In order to evaluate performance and accuracy of the prototype, all measurements from the Internet were also conducted using Nmap, Masscan and ZMap. ZMap and Masscan were running as root because they need access to raw network sockets whereas Nmap and our prototype were running in a regular user context. Each scanner was installed from the Debian repositories and scanning from its own vServer. The server IPs have been whitelisted for scanning before. The versions used are Nmap 7.40, ZMap 2.1.1 and Masscan 1.0.3. Table 5.4 gives an overview of the scanners and how they differ.

Because ZMap is architecturally limited to scan a single port per run, the bash script shown in Listing 17 was iterating the port list and calling ZMap sequentially. Nmap was configured to send no retries, randomize the target host list, skip the ping discovery scan as well as DNS resolutions and to perform an aggressive scan. This way, it behaves similar to the prototype and the other scanners that do not send retries, ping packets or try to resolve DNS. Listing 18 shows the exact command used to invoke Nmap, same for Listing 19 and Masscan respectively. According to Masscan documentation, a virtual machine running Linux is capable to do 300.000 packets per second, so we set this value as Masscan's scan rate because otherwise it would only send 100 packets per second which is way too slow. ZMap seems to have accurate logic for detecting the line speed, so no explicit sending speed configuration is done. The prototype was configured with no rate limit, 1000 concurrent workers and a connection timeout of 2000ms. Also, higher layer application scanning was enabled for HTTP and SSH. The scan was performed using a single node.

Table 5.5 shows the complete results of all our scans whereas Figure 5.5 gives an overview of how many open ports were found in total by the scanners over all measurements. The scan times of all measurements can be found in Table 5.6 and Table 5.7 gives an intuitive overview of scan speeds by showing the respective average of ports per minute found by each scanner per scan. The reason for ZMap being significantly slower than Masscan is that Masscan supports scanning multiple ports in a single run whereas for ZMap, each scanned port requires a dedicated scan, also including an additional cool down time which is eight seconds in the default configuration.

According to our measurements, Nmap was able to find the most open ports, 1840, followed by Masscan, ZMap and the prototype. The most ports were found by Nmap during M1 whereas the prototype discovered 1815 ports in the same measurement which is also the lowest number of ports found across all scans. This means that in the measurement with the highest discrepancy, the prototype found 1.4% less ports than Nmap but taking only 8.31% the time Nmap needed for the same scan. The highest discrepancies between Nmap results and systems found by the prototype are SSH. There seem to be some moving systems that have not been online when the scan started

	Nmap			ZMap			Masscan			Prototype		
	M1	M2	M3	M1	M2	M3	M1	M2	M3	M1	M2	M3
21	12	12	11	12	12	11	12	12	11	12	12	11
22	541	536	530	525	530	524	529	531	524	523	531	524
23	1	1	0	1	1	0	1	1	0	1	1	0
25	5	4	4	5	4	4	5	4	4	5	4	4
53	9	8	8	8	8	8	8	8	8	8	8	8
80	509	516	518	509	516	518	509	516	518	509	514	515
81	3	2	2	3	2	2	3	2	2	3	2	2
110	6	7	7	6	7	7	6	7	7	6	7	7
111	42	42	45	42	42	45	42	42	45	42	42	45
113	2	2	2	2	2	2	2	2	2	2	2	2
135	1	1	0	1	1	0	1	1	0	1	1	0
143	13	14	14	13	14	14	13	14	14	13	14	14
179	2	2	2	2	2	2	2	2	2	2	2	2
443	397	401	403	396	399	402	397	402	405	394	400	401
445	0	1	0	0	1	0	0	1	0	0	1	0
465	25	24	24	25	24	24	25	24	24	25	24	24
515	2	2	2	2	2	2	2	2	2	1	2	2
548	1	2	2	1	2	2	1	2	2	1	2	2
587	23	22	22	23	22	22	23	22	22	23	22	22
646	2	2	2	2	2	2	2	2	2	2	2	2
993	18	18	18	17	18	18	17	18	18	17	18	18
995	10	10	10	9	10	10	9	10	10	9	10	10
1433	1	1	0	1	1	0	1	1	0	1	1	0
1720	0	0	2	0	0	2	0	0	2	0	0	2
1723	5	5	4	5	5	4	5	5	4	5	5	4
2000	6	6	6	6	6	6	6	6	6	6	6	6
3306	12	12	11	12	12	11	12	12	11	12	12	11
3389	53	52	49	53	52	49	53	52	49	53	51	49
5060	2	2	1	2	2	1	2	2	1	2	2	1
5900	2	2	2	2	2	2	2	2	2	2	2	2
6001	1	1	1	1	1	1	1	1	1	1	1	1
8000	93	90	89	93	90	89	93	90	89	93	90	89
8080	31	27	27	31	27	27	31	27	27	31	27	27
8443	2	2	2	2	2	2	2	2	2	2	2	2
8888	2	1	1	2	1	1	2	1	1	2	1	1
10000	1	1	1	1	1	1	1	1	1	1	1	1
49152	1	1	1	1	1	1	1	1	1	1	1	1
49154	4	4	4	4	4	4	4	4	4	4	4	4

Table 5.5: Ports discovered by scanning from the Internet. Ports always found to be closed are omitted.

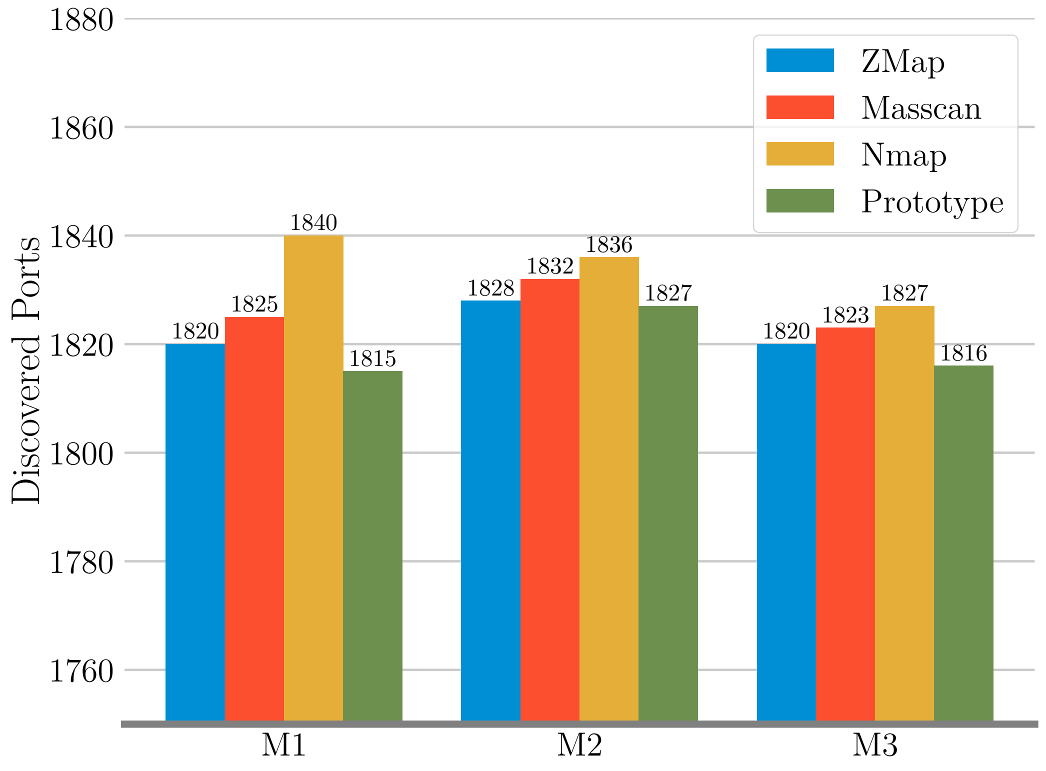

Figure 5.5: Number of ports discovered by scanning from the Internet with ZMap, Masscan, Nmap and the prototype in three different measurements.

because for HTTP and HTTPS the numbers do not differ that much. The results regarding SSH also differ between Nmap and ZMap/Masscan, therefore systems showing up in the morning while Nmap was still running and the other scanners were already finished seem to be the most plausible explanation.

The numbers in Table 5.7 show that the prototype especially offers improvements in situations where asynchronous scanners like ZMap or Masscan cannot be used, for example on when scanning from systems with standard user permissions. The results are comparable to all other scanners, showing that user space port scanning has far more potential than is leveraged by Nmap.

5.2.3 Multiple scan nodes on a single host

During the measurements we decided to run another series of tests, comparing the performance of the prototype when multiple nodes are started on the same machine. The prototype was built to run in user space and to allow horizontal scaling by deploying multiple scanner nodes. Under Linux, the limiting factor for a TCP connect scan in user

Scanner	M1	M2	M3
Nmap	1166m16.330s	1140m21.640s	1201m32.521s
ZMap	7m22.056s	7m22.171s	7m24.316s
Masscan	0m47.282s	0m46.328s	0m46.280s
Prototype	96m54.420s	97m01.517s	96m58.155s

Table 5.6: Scan durations.

Scanner	M1	M2	M3
Nmap	1.578	1.61	1.521
ZMap	247.03	248.05	245.77
Masscan	2315.89	2372.65	2363.44
Prototype	18.73	18.83	18.73

Table 5.7: Discovered ports per minute.

space is the maximum number of file descriptors a process may have, which is 1024 on most default Linux configurations for a standard user. For this measurement, up to 35 nodes were spawned with an one second offset on one of our test virtual machines, jointly performing the scan using the same options used for scans described above (2000ms timeout, 1000 workers per process). With 35 nodes running, both virtual CPUs reached 100% for most of the scan duration whereas the RAM consumption stayed under 1024 Mbyte for the whole system.

Figure 5.6 shows the scan durations and with 34 nodes, a scan against the 53248 target hosts on 50 ports was performed in 3 minutes and 15 seconds. Unfortunately, the scan results were also affected by the high number of scans. Whereas the single node found 1810 ports to be open, 15 found 1777 and 35 nodes were only able to detect 1768 open ports. The complete result overview is shown in Table 5.8. One possible explanation may be that the scanner IP got blacklisted by local protection mechanisms like fail2ban but the exact reasons for this drop are unclear and need further investigation in the future. Nevertheless, depending on the requirements, running multiple scanner nodes on the same machine allows to find a good trade off between scan speed and result completeness.

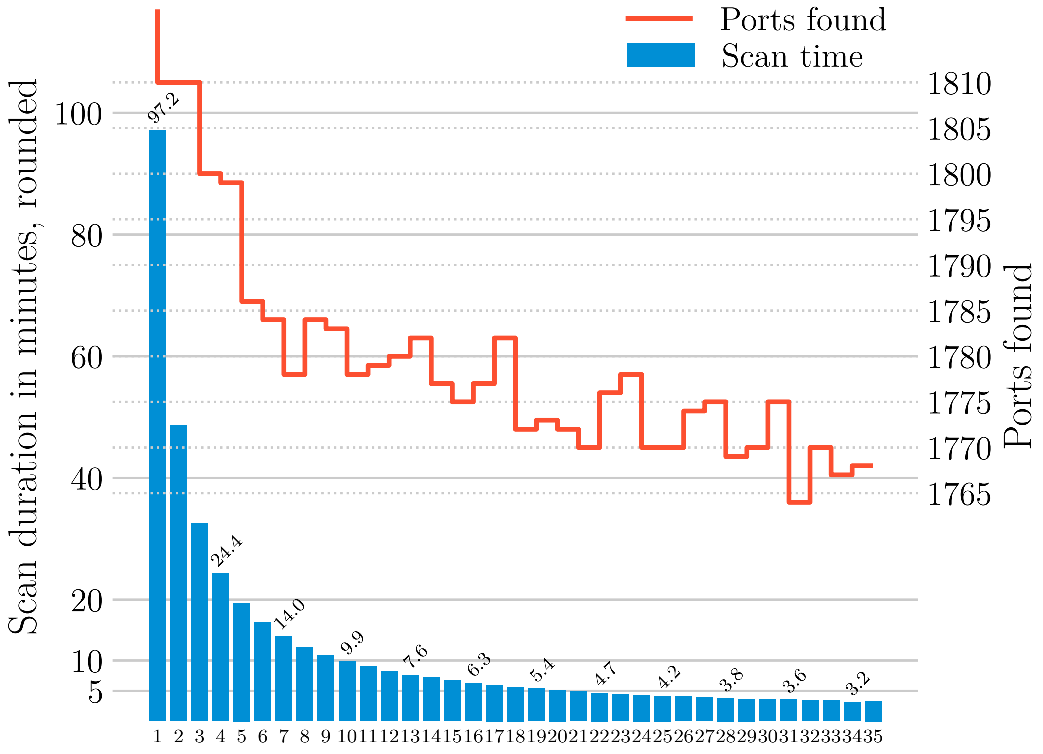

Figure 5.6: Scan duration in relation to the number of nodes running in parallel on a single scan system.

Nodes	Ports found	Runtime (sec)	Ports/Minute
1	1818	5832.258	18.70
2	1810	2914.331	37.26
3	1810	1947.559	55.76
4	1800	1463.076	73.82
5	1799	1170.470	92.22
6	1786	976.871	109.70
7	1784	841.447	127.21
8	1778	733.332	145.47
9	1784	655.478	163.30
10	1783	591.479	180.87
11	1778	540.098	197.52
12	1779	494.386	215.90
13	1780	454.308	235.08
14	1782	429.944	248.68
15	1777	401.470	265.57
16	1775	376.480	282.88
17	1777	358.296	297.58
18	1782	335.450	318.74
19	1772	321.325	330.88
20	1773	306.982	346.54
21	1772	292.318	363.71
22	1770	279.558	379.89
23	1776	271.523	392.45
24	1778	259.580	410.97
25	1770	253.503	418.93
26	1770	245.056	433.37
27	1774	232.865	457.09
28	1775	226.984	469.20
29	1769	221.282	479.66
30	1770	216.833	489.78
31	1775	214.060	497.52
32	1764	204.512	517.52
33	1770	202.665	524.02
34	1767	194.388	545.40
35	1768	196.298	540.40

Table 5.8: Ports found, runtime and ports per minute when running multiple nodes on a single machine.

Chapter 6

Conclusion

6.1 Contributions

In this work we presented a prototype implementation that allows to distribute a port scan across scanner nodes. By using the Go programming language and standard operating system interfaces, it is possible to create fully static, platform independent binaries for most operating systems and platforms in use nowadays, allowing easy and flexible deployment on existing and heterogeneous infrastructure even without elevated user permissions. The server-client design with a pull based approach simplifies communication and node discovery compared to all previous, push-based approaches. The prototype can be extensively configured using a configuration file, supports TCP, UDP, HTTP(S) and SSH scanning and results are event based, enabling work on preliminary results during a scan with a wide variety of tools like databases, log and event processing tools or from the command line. The prototype is reasonably fast, during our measurements more than 10 times faster than Nmap, and its accuracy is comparable to other widespread scanners. Measurements in a testbed show that the scanner is able to sufficiently deal with packet loss which is especially a problem when using asynchronous scanners without retransmissions. Our measurement results show that it is possible to increase scan speed nearly linearly by distributing the scan across multiple nodes. Scan distribution also allows to gain additional insights by generating network views from different vantage points, for example in order to draw conclusions on firewall configuration. Integration of higher layer protocol scanning allows to efficiently extract information from widely used protocols, e.g. all relevant data on TLS certificates can be searched and analyzed without conducting a supplemental scan.

6.2 Future Work

There are several ideas for improving the prototype in the future. First, implementation of the complete ZGrab2 feature set allows to perform in depth scans of many more protocols and would be a useful feature for most users. Apart from ZGrab2 functionality, supporting widespread protocols allows to perform more efficient information gathering in a single scan. A feature distinguishing it further from other scanners would be the ability to load targets from different data sources than files, for example extraction of target domain names from certificate SANs or certificate transparency logs, but also from DNS zone transfers, systems like directory services (LDAP, Active Directory) or asset management systems.

The prototype has large potential for orchestrating scans across fleets of nodes. There are many possible improvements in this space, from registering and assigning multiple scans to different pools to dynamic node management. Also, such management functionality may be exposed by an API which allows creation of language bindings or graphical management interfaces. An included, simple web server providing binaries ready to use may ease deployment.

When it comes to scanning, there may be more known protocols for UDP scanning and functionality like pausing and resuming a complete scan or sending retries. Also, timeout logic is currently straightforward and leaves space for improvements, e.g. some ideas presented in [24] may also be applicable to improve scan performance. Additional functionality like traversing proxy servers may also improve observation capabilities of scanner nodes. A completely passive scan that only observes network traffic in order to discover services may be interesting for sensitive networks.

Chapter A

Appendix

```python
from mininet.net import Mininet
from mininet.log import setLogLevel, info
from mininet.topo import Topo
from mininet.node import Node
from mininet.link import TCLink
from mininet.cli import CLI


''' Adapted from https://github.com/mininet/mininet/wiki/Introduction-to-Mininet
    and https://github.com/mininet/mininet/blob/master/examples/linuxrouter.py '''

WIN_CLIENT = set([445, 139])
LINUX_SRV = set([22])
WIN_SRV = set([3389])
DC_SRV = set([88, 389, 445, 636])
WEB_SRV = set([80, 443])
MAIL_SRV = set([25, 110, 143, 465, 587, 993, 995])
FTP_SRV = set([20, 21])
MYSQL_SRV = set([3306])
WEB_DEV = set([8080])
MYNET = {
    "h0": WIN_CLIENT.union(WIN_SRV, WEB_DEV),
    "h1": WIN_CLIENT,
    "h2": WIN_CLIENT,
    "h3": WIN_CLIENT,
    "h4": WIN_CLIENT,
    "h5": WIN_CLIENT,
    "h6": WIN_CLIENT,
    "h7": WIN_CLIENT.union(WEB_DEV),
    "h8": WIN_CLIENT,
    "h9": WIN_SRV.union(DC_SRV),
    "h10": WIN_CLIENT,
    "h11": WIN_CLIENT.union(WIN_SRV),
    "h12": WIN_CLIENT,
    "h13": WIN_CLIENT,
    "h14": WIN_CLIENT,
    "h15": WIN_CLIENT,
    "h16": WIN_CLIENT,
    "h17": WIN_CLIENT,
    "h18": WIN_CLIENT.union(WIN_SRV),
    "h19": WIN_SRV.union(DC_SRV),
    "h20": WIN_CLIENT,
    "h21": WIN_CLIENT.union([8000]),
    "h22": WIN_CLIENT,
    "h23": WIN_CLIENT,
    "h24": WIN_CLIENT,
    "h25": WIN_CLIENT,
    "h26": WIN_CLIENT,
    "h27": WIN_CLIENT.union(MYSQL_SRV),
    "h28": WIN_CLIENT,
    "h29": WIN_CLIENT,
    "h30": WIN_CLIENT,
    "h31": WIN_CLIENT.union(MYSQL_SRV),
    "h32": WIN_CLIENT,
    "srv0": LINUX_SRV.union(WEB_SRV),
    "srv1": LINUX_SRV.union(WEB_SRV),
    "srv2": LINUX_SRV.union(WEB_SRV, MYSQL_SRV),
    "srv3": LINUX_SRV.union(WEB_SRV),
    "srv4": LINUX_SRV.union(WEB_SRV, FTP_SRV),
    "srv5": WIN_SRV.union(WEB_SRV, MAIL_SRV),
    "srv6": LINUX_SRV.union(WEB_SRV, set([8000, 8443])),
    "srv7": LINUX_SRV.union(WEB_SRV),
    "srv8": WIN_SRV.union(WEB_SRV),
    "srv9": LINUX_SRV.union(WEB_SRV),
    "srv10": LINUX_SRV.union(FTP_SRV),
    "srv11": LINUX_SRV.union(MAIL_SRV),
    "srv12": LINUX_SRV.union(WEB_SRV, FTP_SRV),
    "srv13": WIN_SRV.union(WEB_SRV),
    "srv14": LINUX_SRV.union(WEB_SRV, MYSQL_SRV),
    "srv15": LINUX_SRV.union(FTP_SRV),
    "srv16": LINUX_SRV.union(WEB_SRV, MAIL_SRV),
    "srv17": LINUX_SRV.union(WEB_SRV),
    "srv18": WIN_SRV.union(MAIL_SRV),
}

def runService(node, port):
    ''' sudo apt install nmap
        ncat -lkv 0.0.0.0 <port>
    '''
    node.cmdPrint("ncat -lkv 0.0.0.0 %s &" % port)


class Router(Node):
    "A Node with IP forwarding enabled."

    def config(self, **params):
        super(Router, self).config(**params)
        # Enable forwarding on the router
        self.cmd('sysctl net.ipv4.ip_forward=1')
        self.cmdPrint("iptables-restore < rules.v4")

    def terminate(self):
        self.cmd('sysctl net.ipv4.ip_forward=0')
        super(Router, self).terminate()


class TestTopo(Topo):
    def build(self, n=2, numHosts=99):
        defaultIP = '192.168.1.1/24'  # IP address for r0-eth0
        router = self.addNode('r0', cls=Router, ip=defaultIP)

        s1, s2 = [self.addSwitch(s) for s in ('s1', 's2')]

        # Limit bandwidth and introduce packet loss
        # 10 Mbps, 5ms delay, 2% loss, 1000 packet queue
        # self.addLink(host, switch, bw=10, delay='5ms', loss=2,
        #   max_queue_size=1000, use_htb=True )

        # local network
        linkopts = dict(bw=10, loss=1, max_queue_size=1000, use_htb=True)
        self.addLink(s1, router, intfName2='eth0', params2={'ip':'192.168.1.1/24'},
        ↪  **linkopts)
        # Internet
        self.addLink(s2, router, intfName2='eth1', params2={'ip':'172.16.1.1/24'})

        host_ctr = 0
        # Hosts in local network
        for i in range(0, numHosts):
            h = self.addHost('h%d' % host_ctr, ip='192.168.1.1%02d/24' % i,
                defaultRoute='via 192.168.1.1')
            self.addLink(h, s1)
            host_ctr += 1
        # Host in Internet
        host_ctr = 0
        for i in range(0, numHosts):
            h = self.addHost('srv%d' % host_ctr, ip='172.16.1.1%02d/24' % i,
                defaultRoute='via 172.16.1.1')
            self.addLink(h, s2)
            host_ctr += 1


def main():
    setLogLevel('info')
    test_topo = TestTopo(n=4)
    net = Mininet(topo=test_topo, link=TCLink )
    net.start()
    for host in MYNET:
        for service in MYNET[host]:
            runService(net.get(host), service)
    info( '*** Routing Table on Router:\n' )
    info( net[ 'r0' ].cmd( 'route' ) )
    CLI( net )
    net.stop()


if __name__ == '__main__':
    main()
```

Listing 8: Mininet setup script.

```
# Firewall rules for router in test setup
# eth0 is the local network
# eth1 is the Internet

*filter
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]

# Rules for controlling router

## Ping
-A INPUT -p icmp -j ACCEPT
## Every existing connection
-A INPUT -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
## Local traffic
-A INPUT -i lo -j ACCEPT

## Basically drop everything
-A INPUT -p tcp -j DROP
-A INPUT -p udp -j DROP
-A INPUT -j DROP


# Rules for forwarding
## Every established connection
-A FORWARD -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
## Every new connection from inside the local network going to the Internet
-A FORWARD -i eth0 -m conntrack --ctstate NEW -j ACCEPT
# Allow ping
-A FORWARD -i eth0 -p icmp --icmp-type 8 -m state --state NEW,ESTABLISHED,RELATED -j
↪  ACCEPT


# Open a tcp port X on server Y to the Internet
#-A FORWARD -i eth1 -o eth0 -p tcp --dport X -d Y -m conntrack --ctstate NEW -j ACCEPT

# Internal jump and dev server, reachable from outside
-A FORWARD -i eth1 -p icmp --icmp-type 8 -d 192.168.1.100 -m state --state
↪  NEW,ESTABLISHED,RELATED -j ACCEPT
-A FORWARD -i eth1 -o eth0 -p tcp --dport 3389 -d 192.168.1.100 -m conntrack --ctstate
↪  NEW -j ACCEPT
-A FORWARD -i eth1 -o eth0 -p tcp --dport 8080 -d 192.168.1.100 -m conntrack --ctstate
↪  NEW -j ACCEPT

# Test server, not pingable, but tcp/8000 reachable
-A FORWARD -i eth1 -o eth0 -p tcp --dport 8000 -d 192.168.1.121 -m conntrack --ctstate
↪  NEW -j ACCEPT

# Web dev server, not pingable, but tcp/8080 reachable
-A FORWARD -i eth1 -o eth0 -p tcp --dport 8080 -d 192.168.1.107 -m conntrack --ctstate
↪  NEW -j ACCEPT

COMMIT
```

Listing 9: Firewall rules on testbed router.

```
/* Event is a container for everything that happens
   at a node and should later on be handled by EventHandlers */
message Event {
    string nodeID = 1;
    string nodeName = 2;
    //int32 pool = 3;
    google.protobuf.Timestamp timestamp = 4;
    string scannername = 5;
    oneof EventData {
        EnvironmentInformation environment = 6;
        PortScanResult portscan = 7;
        ZGrab2ScanResult zgrabscan = 8;
    }
}

/* EnvironmentInformation tells the server
   under which circumstances nodes are running */
message EnvironmentInformation {
    string hostname = 1;
    string os = 2;
    string pid = 3;
    string processname = 4;
    string username = 5;
    string cpumodelname = 6;
   }

/* TCPScanResult contains the outcome of
   a TCP scan against a single port on a single host */
message PortScanResult {
    string target = 1;
    uint32 port = 2;
    bool open = 3;
    string scantype = 4;
    uint32 timeout = 5;
}

message ZGrab2ScanResult {
    string target = 1;
    uint32 port = 2;
    google.protobuf.Value jsonResult = 3;
}
```

Listing 10: Event Messages defined for our prototype

```
/* Everything a server sends is packed in a server message */
message ServerMessage {
    oneof MessageContent {
        RegisteredNode registeredNode = 1;
        MoreWorkReply jobBatch = 2;
        HeartbeatAck heartbeatAck = 3;
        WorkDoneAck workDoneAck = 4;
        GoodbyeAck goodbyeAck = 5;
        Unregistered nodeIsUnregistered = 6;
    }
}

/* Everything the nodes sends is packed in a node message */
message NodeMessage {
    oneof MessageContent {
        NodeRegister nodeRegister = 1;
        Heartbeat heartbeat = 2;
        MoreWorkRequest moreWork = 3;
        WorkDone workDone = 4;
        Goodbye goodbye = 5;
    }
}

/* ScanTarget may be a dnsName/ip and a port. ip is encoded as byte array.
   dnsName is a FQDN and mustn't contain a protocol specification */
message ScanTarget {
    string url = 1;
    repeated uint32 tcpports = 2;
    repeated uint32 udpports = 3;
}

/* This message is sent every time a node registers at
   the server. It contains a unique node ID so there are
   not multiple scanner nodes running on the same machine */
message NodeRegister {
    string machineID = 1;
    int32 preferredPool = 2;
    string preferredNodeName = 3;
    Event envinfo = 4;
}

/* This message is sent by the server and indicates that
   the server does not know this node and that the node should
   register again */
message Unregistered {
    string nodeID = 1;
}

/* This message tells the scanner that it is registered at
   the server and assigns a unique scanner ID as well as it carries
   the timestamp of the server, so nodes that have no correct clock
   can still perform somewhat accurate timestamping of events */
message RegisteredNode {
    string NodeID = 1;
    google.protobuf.Timestamp ServerClock = 2;
    bytes scannerconfig = 3;
}
```

Listing 11: Messages defined for our prototype

```
/* A heartbeat message that is sent regularly from any node
   to the server to signal that it is still alive */
message Heartbeat {
    string NodeID = 1;
    google.protobuf.Timestamp BeatTime = 2;
}

/* Acknowledgement of a heartbeat message. A server        may indicate
   to stop scanning and if the scanner should exit */
message HeartbeatAck {
    bool Scanning = 1;
    bool Running = 2;
}

/* Sent by a node if it has no work */
message MoreWorkRequest {
    string NodeID = 1;
}

/* Contains more work for a scanner */
message MoreWorkReply {
    uint64 batchid = 1;
    repeated ScanTarget targets = 3;
}

/* Indicates that a node is done with a work batch
   and contains the results */
message WorkDone {
    string nodeID = 1;
    uint64 batchid = 2;
    repeated Event events = 3;
}

message WorkDoneAck {}
/* Node is going to exit */
message Goodbye {
    string nodeID = 1;
}

/* Server ACKs the goodbye and signals if the node is allowed to exit */
message GoodbyeAck {
    bool ok = 1;
}
```

Listing 12: Messages defined for our prototype

```
jq '. | select(.scannername == "zgrab2-http" and
↪  .zgrabscan.jsonResult.response.request.tls_log != null
↪  and(.zgrabscan.jsonResult.response.request.tls_log.handshake_log⌟
↪  .server_certificates.certificate.parsed.issuer_dn != "C=DE, O=Verein zur Foerderung
↪  eines Deutschen Forschungsnetzes e. V., OU=DFN-PKI,CN=DFN-Verein Global Issuing
↪  CA") and .zgrabscan.jsonResult.response.request.tls_log.handshake_log⌟
↪  .server_certificates.certificate.parsed.issuer_dn != "C=DE, O=Technische
↪  Universitaet Muenchen,CN=Zertifizierungsstelle der TUM" and
↪  .zgrabscan.jsonResult.response.request.tls_log.handshake_log.server_certificates⌟
↪  .certificate.parsed.issuer_dn != "C=US, O=Let\'s Encrypt, CN=Let\'s
↪  EncryptAuthority X3" and .zgrabscan.jsonResult.response.request.tls_log⌟
↪  .handshake_log.server_certificates.certificate.parsed.issuer_dn != "C=DE,
↪  ST=Bayern, L=Muenchen, O=Leibniz-Rechenzentrum, OU=LRZ-CA,CN=LRZ-CA - G01,
↪  emailAddress=pki@lrz-muenchen.de") | .zgrabscan.jsonResult.response.request⌟
↪  .tls_log.handshake_log.server_certificates.certificate.parsed.fingerprint_sha256'
↪  <filename> | uniq | wc -l
```

Listing 13: jq command to show the number of certificates issued by foreign CAs.

```
jq '. | select(.scannername == "zgrab2-http" and
↪  .zgrabscan.jsonResult.response.request.tls_log != null and
↪  (.zgrabscan.jsonResult.response.request.tls_log.handshake_log.server_certificates⌟
↪  .certificate.parsed.validity.end | fromdateiso8601) < now ) |
↪  .zgrabscan.jsonResult.response.request.tls_log.handshake_log.server_certificates⌟
↪  .certificate.parsed.fingerprint_sha256 + ":" +
↪  .zgrabscan.jsonResult.response.request.tls_log.handshake_log.server_certificates⌟
↪  .certificate.parsed.validity.end' <filename> | uniq | wc
↪  -l
```

Listing 14: jq command to show the number of expired TLS certificates.

```
jq '. | select(.scannername == "zgrab2-http" and
↪  .zgrabscan.jsonResult.response.request.tls_log != null) |
↪  .zgrabscan.jsonResult.response.request.tls_log.handshake_log.server_certificates⌟
↪  .certificate.parsed.fingerprint_sha256 + "," +
↪  .zgrabscan.jsonResult.response.request.tls_log.handshake_log.server_certificates⌟
↪  .certificate.parsed.validity.start + "," +
↪  .zgrabscan.jsonResult.response.request.tls_log.handshake_log.server_certificates⌟
↪  .certificate.parsed.validity.end'
↪  <filename>
```

Listing 15: jq command to show sha256 fingerprint, start and end date of all TLS certificates.

```
jq '. | select(.scannername == "zgrab2-http" and
↪  .zgrabscan.jsonResult.response.request.tls_log != null) | .zgrabscan.target + ":" +
↪  (.zgrabscan.port|tostring) + ":" .zgrabscan.jsonResult.response.request.tls_log⌟
↪  .handshake_log.server_certificates.certificate.parsed.issuer_dn + ":"
↪  (.zgrabscan.jsonResult.response.request.tls_log.handshake_log.server_certificates⌟
↪  .certificate.parsed.names | length | tostring)' <filename> | wc
↪  -l
```

Listing 16: jq command to show the certificate issuer and number of SANs in certificates provided by a given host:port combination.

```
#!/bin/bash

ports=(80 23 443 21 22 25 3389 110 445 139 143 53 135 3306 8080 1723 111 995 993 5900
↪  1025 587 8888 199 1720 465 548 113 81 6001 10000 514 5060 179 1026 2000 8443 8000
↪  32768 554 26 1433 49152 2001 515 8008 49154 1027 5666 646)

for port in "${ports[@]}"
do
        echo "Scanning $port at `date`"
        zmap -p $port 131.159.0.0/16 --blacklist-file ./rbg-excludes.txt -o
        ↪  "scan-$port.zmap" -q
        echo ""
done
```

Listing 17: Bash script used to invoke Zmap for all target ports.

```
nmap -v -T4 -Pn -n --max-retries 0 --randomize-hosts --excludefile ./rbg-excludes.txt
↪  --top-ports 50 -oA scan.nmap 131.159.0.0/16
```

Listing 18: Command used to invoke Nmap for the evaluation.

```
masscan --range 131.159.0.0/16 --excludefile ./rbg-excludes.txt --ports
↪  80,23,443,21,22,25,3389,110,445,139,143,53,135,3306,8080,1723,111,995,993,5900,⌟
↪  1025,587,8888,199,1720,465,548,113,81,6001,10000,514,5060,179,1026,2000,8443,8000,⌟
↪  32768,554,26,1433,49152,2001,515,8008,49154,1027,5666,646 --rate 300000 -oB
↪  scan.masscan
```

Listing 19: Command used to invoke Masscan for the evaluation.

Bibliography

[1] *A distributed nmap / masscan scanning framework.* URL: `https://github.com/rackerlabs/scantron` (visited on 11/30/2018).

[2] David Adrian et al. „Zippier ZMap: Internet-Wide Scanning at 10 Gbps.“ In: *WOOT*. 2014.

[3] Neal Cardwell and Yuchung Cheng. „BBR congestion control“. In: *Working Draft, IETF Secretariat, Internet-Draft draft-card-well-iccrg-bbr-congestion-control-00* (2017).

[4] *Censys.* URL: `https://censys.io/` (visited on 12/17/2018).

[5] *Data Race Detector.* URL: `https://golang.org/doc/articles/race_detector.html` (visited on 03/21/2019).

[6] *DFN PKI.* URL: `https://www.pki.dfn.de/ueberblick-dfn-pki/` (visited on 03/23/2019).

[7] *DNMAP.* URL: `https://sourceforge.net/p/dnmap/wiki/Home/` (visited on 11/30/2018).

[8] Zakir Durumeric, Eric Wustrow, and J Alex Halderman. „ZMap: Fast Internet-wide scanning and its security applications“. In: *Proceedings of the 22nd USENIX Security Symposium.* 2013.

[9] Zakir Durumeric et al. „A search engine backed by Internet-wide scanning“. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* ACM. 2015, pp. 542–553.

[10] *Free Pool of IPv4 Address Space Depleted.* URL: `https://www.nro.net/ipv4-free-pool-depleted` (visited on 03/21/2019).

[11] Fyodor. *The Art of Port Scanning.* URL: `http://phrack.org/issues/51/11.html` (visited on 11/18/2018).

[12] *Go syslist - List of archtiectures and operating systems supported by Go.* URL: `https://github.com/golang/go/blob/master/src/go/build/syslist.go` (visited on 03/21/2019).

[13] *GoDoc rate - Package rate provides a rate limiter.* URL: `https://godoc.org/golang.org/x/time/rate` (visited on 03/21/2019).

[14] Robert David Graham. *MASSCAN: Mass IP port scanner.* URL: https://github.com/robertdavidgraham/masscan (visited on 11/18/2018).

[15] *Hackability Explorer.* URL: https://srlabs.de/hackability/ (visited on 12/17/2018).

[16] *Internet Census 2012.* URL: http://census2012.sourceforge.net/paper.html (visited on 12/17/2018).

[17] Gordon Lyon. *Nmap: the Network Mapper - Free Security Scanner.* URL: https://nmap.org/ (visited on 11/18/2018).

[18] *Mail thread on dailydave mailing list discussing drone features of unicornscan.* URL: https://seclists.org/dailydave/2004/q4/0 (visited on 12/17/2018).

[19] *Metasploit - auxiliary/scanner/discovery/udp-sweep.* URL: https://github.com/rapid7/metasploit-framework/blob/2bb0d8491f41cdf777dab53dac8b7fff563ef43e/modules/auxiliary/scanner/discovery/udp_sweep.rb (visited on 03/21/2019).

[20] *[mininet-discuss] Routing table bug when using TCLink.* URL: https://mailman.stanford.edu/pipermail/mininet-discuss/2019-March/008184.html (visited on 04/07/2019).

[21] *nanomsg/mangos - mangos-v2 is version 2 of an implementation in pure Go of the SP ("Scalable Protocols") protocols.* URL: https://github.com/nanomsg/mangos (visited on 03/21/2019).

[22] *Nmap Book, Chapter 4. Port Scanning Overview: Top 20 (most commonly open) TCP ports.* URL: https://nmap.org/book/port-scanning.html#most-popular-ports (visited on 03/21/2019).

[23] *Player 3 Has Entered the Game: Say Hello to 'WannaCry'.* URL: https://blog.talosintelligence.com/2017/05/wannacry.html (visited on 03/23/2019).

[24] *Port Scanning improved - New ideas for old practices.* URL: https://media.ccc.de/v/24c3-2131-en-port_scanning_improved#t=0 (visited on 03/27/2019).

[25] J. Postel. *Character Generator Protocol.* RFC 864. 1983, pp. 1–3. URL: https://tools.ietf.org/html/rfc864.

[26] *Project Sonar.* URL: https://www.rapid7.com/research/project-sonar/ (visited on 12/17/2018).

[27] *Protocol Buffers.* URL: https://developers.google.com/protocol-buffers/ (visited on 04/03/2019).

[28] R. van Rijswijk-Deij et al. „A High-Performance, Scalable Infrastructure for Large-Scale Active DNS Measurements“. In: *IEEE Journal on Selected Areas in Communications* 34.6 (June 2016), pp. 1877–1888. ISSN: 0733-8716. DOI: 10.1109/JSAC.2016.2558918.

[29] *RIPE Atlas.* URL: https://atlas.ripe.net/ (visited on 04/03/2019).

[30] *Shodan.* URL: https://www.shodan.io/ (visited on 12/17/2018).

[31] *spf13/cobra - A Commander for modern Go CLI interactions.* URL: https://github.com/spf13/cobra (visited on 03/21/2019).

[32] *spf13/viper - Go configuration with fangs.* URL: https://github.com/spf13/viper (visited on 03/21/2019).

[33] *The Go Memory Model - Channel Communication.* URL: https://golang.org/ref/mem#tmp_7 (visited on 03/21/2019).

[34] *The Go Programming Language.* URL: https://golang.org/ (visited on 03/21/2019).

[35] *The Go Programming Language - Command link.* URL: https://golang.org/cmd/link/ (visited on 03/21/2019).

[36] *Unicornscan Author Jack C. Louis Dies in Tragic fire at his home in Sweden.* URL: http://blogs.hackerscenter.com/2009/03/unicornscan-author-jack-c-louis-dies-in.html (visited on 12/17/2018).

[37] *unicornscan Github Mirror.* URL: https://github.com/dneufeld/unicornscan (visited on 12/17/2018).

[38] *unicornscan(1).* URL: https://linux.die.net/man/1/unicornscan (visited on 12/17/2018).

[39] *Wolpertinger distributed portscanner.* URL: https://github.com/SySS-Research/wolpertinger (visited on 12/17/2018).

[40] *Wolpertinger. Ein verteilter Portscanner - Presentation at 26C3.* URL: https://media.ccc.de/v/26c3-3340-de-wolpertinger_ein_verteilter_portscanner (visited on 12/17/2018).

[41] *Worldwide broadband speed league 2018.* URL: https://www.cable.co.uk/broadband/speed/worldwide-speed-league/ (visited on 03/30/2019).

[42] *ZGrab2 - Go Application Layer Scanner.* URL: https://github.com/zmap/zgrab2 (visited on 03/21/2019).

[43] *zmap/go-iptree - GoLang IP Radix Tree.* URL: https://github.com/zmap/go-iptree (visited on 03/21/2019).