University of Zurich[UZH]

MASTER THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Simulator of Distributed Datasets for Pulse-wave DDoS Attacks

Pascal Kiechl
Zurich, Switzerland
Student ID: 16-927-998

Supervisor: Dr. Bruno Rodrigues, Katharina O. E. Müller, Prof. Dr. Burkhard Stiller
Date of Submission: August 06, 2023

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmuehlestrasse 14, CH-8050 Zurich, Switzerland
URL: http://www.csg.uzh.ch/

Zusammenfassung

Die fortlaufend zunehmende Stärke und Häufigkeit von Distributed Denial-of-Service (DDoS) Angriffen, sowie auch das Aufkommen neuer Angriffsmethoden, wie zum Beispiel Pulse-Wave DDoS Angriffe, betonen wie wichtig es ist, dass Gegenmassnahmen mit der eskalierenden Bedrohung mithalten können. Zu diesem Zweck wurde bereits viel Forschung betrieben, die das Ziel hat, DDoS Datensätze zu generieren, welche die Grundlage für die Entwicklung von Gegenmassnahmen wie Intrusion Detection Systems (IDS) legen.

Allerdings repräsentieren bestehende Datensätze in der Regel einen einzelnen, Opferbasierten Blickwinkel, was gegenüber einem verteilten Datensatz limitiert ist, da dieser zahlreiche verschiedene Perspektiven auf einen Angriff beinhaltet. Daher implementiert diese Arbeit einen Simulator der solche verteilte Datensätze generieren kann. Dabei wird der Fokus auf Pulse-Wave DDoS Angriffe gelegt, da für diese bis anhin keine öffentlich verfügbare Datensätze existieren. Der Simulator ermöglicht die Kreation von verschiedenen Topologien und Angriffs-Zusammensetzungen durch hohe Flexibilität in den Konfigurationsmöglichkeiten.

Die Auswertung zeigt die Fähigkeit des Tools eine grosse Vielfalt von verschiedenen Datensätzen zu erzeugen, die unterschiedliche Eigenschaften aufweisen bezüglich häufig verwendeter DDoS Fingerabdruck-Metriken. Daher repräsentiert diese Arbeit einen bedeutenden Schritt vorwärts in der Erforschung von Pulse-Wave DDoS Angriffen, was die Entwicklung verbesserter Gegenmassnahmen unterstützt.

Abstract

The ever increasing scale and frequency of Distributed Denial-of-Service (DDoS) attacks, as well as the emergence of new forms of attacks, such as pulse-wave DDoS attacks, highlights the importance of ensuring that mitigation capabilities are able to keep up with the escalating threat posed by DDoS attacks. To that end, much work has been done with regard to the generation of DDoS datasets which form the basis for developing effective mitigation tools such as Intrusion Detection Systems (IDS).

However, existing datasets typically represent a single, victim-centric viewpoint, which has limitations compared to a distributed dataset that provides multiple different perspectives onto an attack. Thus, this thesis implements a simulator for distributed datasets specifically focused on pulse-wave DDoS attacks, for which at current no datasets are publicly available. The simulator provides high flexibility and configurability in the types of use cases that can be modeled, allowing for the creation of different topologies and attack compositions.

The evaluation demonstrates the tool's capability to create of a wide range of diverse datasets that exhibit different characteristics with regard to metrics that are commonly used in a DDoS attack's fingerprint. As such, this thesis represents a significant step towards enabling a better understanding of pulse-wave DDoS attacks and thereby the development of improved tools to help defend against them.

Acknowledgments

I wish to express my gratitude to the people who have supported me throughout this thesis. My supervisors Dr. Bruno Rodrigues and Katharina Müller deserve special thanks for their efforts and so does Prof. Dr. Burkhard Stiller to whom I owe the opportunity to work on this project at the Communication Systems Group at the UZH.

Dr. Bruno Rodrigues in particular has provided me with invaluable guidance and support. He has always been available for discussions and has given important feedback throughout the entirety of the project.

Finally, I would like to thank Fabian Küffer for his assistance with proofreading the report.

Contents

Chapter 1

Introduction

Distributed Denial-of-Service (DDoS) attacks represent a persistent threat to network security in today's world, with attacks continuously increasing in terms of complexity and sophistication [64, 65]. In 2017, the term pulse-wave DDoS attack was coined and applied to a new approach to performing DDoS attacks [77]. Pulse-wave DDoS attacks are capable of generating high amounts of traffic with near-immediate ramp up, producing short and repeating bursts of DDoS traffic that have proven detrimental to DDoS mitigation systems, specifically so-called appliance first hybrid mitigation systems [77].

Frequently, an Intrusion Detection System (IDS) is one of the most important systems for ensuring the secure operation of a network and usually represents the first line of defense, receiving and analyzing incoming traffic before the traffic arrives at an Intrusion Prevention System (IPS), such as a firewall appliance [20]. IDSs rely on comparing inbound traffic to known characteristics of previous attacks (fingerprints) to raise alerts on potential attacks [5]. Therefore the generation of network traces to evaluate the effectiveness of an IDS's signatures and to update the system to match previously undetected cases is critical [47].

1.1 Motivation

One limitation of existing DDoS datasets in general is that they frequently focus on the victim, *i.e.,* the target of the attack [1, 5]. While considering the victim's point of view is a relatively simple and efficient approach to understanding and profiling simple attack patterns, such as volumetric attacks, it has a limitation in understanding the origins of systems orchestrating such attacks, such as Botnets [67]. The benefits of abandoning the victim-centric perspective in favor of a distributed view onto an attack are significant, providing a more complete picture of the attack as it evolves on its path towards the victim and ideally enabling the prevention of the attack before it reaches its target through the application of cooperative DDoS mitigation techniques [64].

Thus, this thesis proposes an approach for the generation of such distributed datasets specifically with pulse-wave DDoS attacks mind, thereby filling a gap in existing research.

As such the major contribution of the thesis is the open source prototype system capable of generating a diverse range of distributed pulse-wave datasets that exhibit specific characteristics based on user configuration. The resulting datasets provide the foundational input for the future creation of pulse-wave DDoS attack fingerprints based on a distributed perspective, thus enabling further research and contributing to the development of collaborative DDoS mitigation techniques centered around pulse-wave attacks.

1.2 Thesis Goals

Consequently, the main objective of the thesis is the design and implementation of a prototype system that is capable of producing pulse-wave DDoS traffic traces in form of a distributed dataset.

Furthermore, the prototype must operate in a way that allows the characteristics of the generated dataset to be determined by configurable parameters such as the protocols used in the attack, the overall topology and other factors that are used in determining attack signatures. In this way, the system-to-be shall support the execution of a wide range of use cases and pulse-wave attacks that result in distributed datasets which show characteristics that are indicative of having different attack fingerprints.

Lastly, the thesis must evaluate the proposed system regarding its capabilities to generate diverse, distributed pulse-wave datasets as well as analyse the system's scalability such that its ability to simulate large-scale use cases can be determined.

1.3 Methodology

The methodology applied to this thesis is as follows: As a first step, a literature review is conducted with the goal of obtaining the necessary theoretical foundation needed to build such a simulator. This review naturally includes the topic of pulse-wave DDoS attacks, but also covers dataset generation in the realm of DDoS traffic analysis, as well as general fundamental knowledge about DDoS attacks.

This is followed by designing and implementing the system, making use of the insights acquired in the literature review. The phases of design and implementation are somewhat less clearly separated, with learnings from implementing being fed back into the design, thus lending a somewhat exploratory character to this part of the work.

The evaluation represents the last link in the chain of activities, with focus being put on demonstrating the prototype implementation's capabilities regarding the generation of a wide range of different pulse-wave patterns and attack vector characteristics. A second point of focus is the analysis of how the system performs and scales. Though achieving scalability is not a direct goal of this thesis, insights gained from analyzing the system's capabilities in this regard are still is relevant to determine the development path of the prototype going forward.

1.4 Thesis Outline

The rest of the thesis is structured as follows: In Chapter 2 the required foundational knowledge regarding DDoS, including pulse-wave DDoS and dataset generation is established. Further, existing work in the realms of dataset generation and pulse-wave DDoS attacks is examined and contrasted against this thesis. The design of the system-to-be is described in Chapter 3 and the subsequent implementation is documented in Chapter 4. Then, the system is evaluated and the findings are discussed in Chapter 5. Finally, in Chapter 6, the results of this work are summarized, conclusions are drawn and possible future work is outlined.

Chapter 2

Fundamentals

The purpose of this chapter is twofold, as it aims to lay the theoretical groundwork needed for the proper understanding of the discussed material and provide the relevant practical and research context in which this work is situated.

Thus, in the background section, the theoretical basis is established, whilst the related work section discusses prior work done in the realm of Distributed Denial-of-Service (DDoS) attack dataset generation and existing literature regarding pulse-wave DDoS attacks.

2.1 Background

In this part of the thesis, the concepts surrounding DDoS, including a coarse-grained categorization of attack types and the possible motivation of the attackers are explained. Further, pulse-wave DDoS attacks are differentiated from 'regular' DDoS attacks and the topic of DDoS attack datasets, the role they play in research, their availability and the ways they can be generated are discussed.

2.1.1 Distributed Denial-of-Service (DDoS)

Fundamentally speaking, a Denial-of-Service (DoS) attack aims to disrupt the use (*i.e.,* service) of a network resource [8, 46]. If successful, such an attack usually either makes the targeted service completely unavailable or only available at lower performance, causing harm to both service providers and service users alike [8].

DDoS attacks constitute a logical step in the evolution of DoS attacks, in which not just one but multiple attacking entities are used in combination to achieve service denial [46]. The distributed nature of DDoS attacks makes them both more potent in terms of *e.g.,* the traffic volume it can generate and more difficult to defend against when compared to their non-distributed counterpart [24].

2.1.1.1 Incentives for Attackers

The reasons for executing a DDoS attack vary greatly, as do the targets of the attacks, ranging from government institutions, over commercial and political targets, to private individuals [45, 46]. [12] outline how the practice of performing DDoS attacks has spread over time from 'hacker culture' for the purpose of social protest and pranks to being adopted by organized crime, nation states and governments for financial gains and control.

Five categories of motivations for conducting DDoS attacks are put forth by [45]:

- **Financial or economic benefit**: Attacks carried out in pursuit of monetary gains are typically performed by experienced technicians. Thus, considered highly dangerous and difficult to stop.
- **Revenge**: Typically, attacks in this category are conducted by lower-skilled individuals who feel slighted or oppressed.
- **Ideological belief**: Ideologically motivated attacks are not as frequent but have historically made up some of the most highlighted DDoS events, such as *e.g.,* the DDoS attack on WikiLeaks in 2010.
- **Intellectual challenge**: Attacks falling under this category serve the purpose of demonstrating the attackers' skills and capabilities. Readily available and user-friendly tools constitute an important facilitating factor for this category.
- **Cyberwarfare**: This category contains attacks perpetrated on countries by other countries or terrorist organizations, typically conducted by skilled people with significant resources. Consequently, the impact on the target's infrastructure and economy is potentially debilitating.

2.1.1.2 Categories of DDoS Attacks

DDoS attacks do not just vary in terms of motivation and the types of attackers and targets, as outlined in Section 2.1.1.1, but also in terms of their technical makeup [8, 45, 46]. Crucially, the tools and mechanisms used by attackers continue to increase in sophistication, and new options are continuously being explored to circumvent existing defensive measures [8, 45, 46].

Given the ever-evolving nature of DDoS attack mechanisms, taxonomies and classifications (*cf.* taxonomies put forth by [45, 46]) of attack methods will need to be continuously refined. The aforementioned taxonomies represent powerful and detailed means for the classification of attacks, but for the purpose of this work, it is sufficient to maintain a comparatively coarse-grained categorization of DDoS attacks into two major categories, which represent high-level categories of the taxonomy established by [8]:

- **Application Layer Attack** ('Resource Exhaustion Attack' in [45]): DDoS attacks falling under this category aim to exhaust the resources of the host system (*i.e.,*

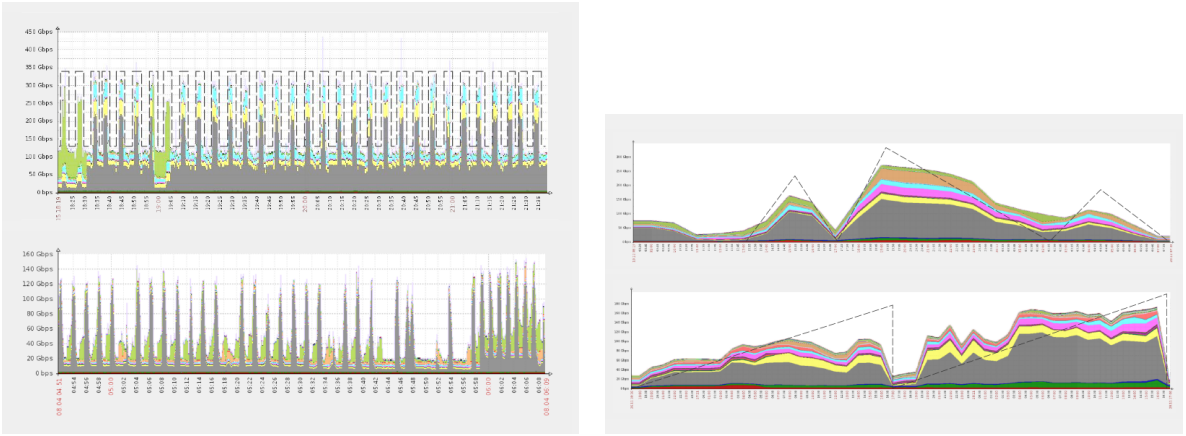
server), such as memory, CPU, disk bandwidth and sockets to ultimately render the system unresponsive or make it crash completely [8, 45].

- **Network Layer Attack** ('Bandwidth Depletion Attack' in [45]): This group of attacks targets the victim's network bandwidth or routing capacity, thus rendering it incapable of processing legitimate user's service requests [8, 45]. In that way, bandwidth depletion attacks undermine the victim's connectivity, rather than directly targeting its ability to provide its services, the way a resource depletion attack would [8].

2.1.1.3 Pulse-Wave DDoS Attacks

A pulse-wave attack is a new mechanism for orchestrating DDoS attacks that has emerged relatively recently [3, 16]. This attack mechanism is characterized by consisting of 'pulses' of short-duration but high-rate traffic, making it different from more traditional forms of DDoS attacks, which consist of traffic that ramps up much slower, but then is maintained for longer periods of time [3, 16].

To exemplify that difference visually, Figure 2.1a shows two examples of pulse-wave DDoS attacks with their characteristic traffic spikes (*i.e.,* pulses), whereas the conventional way of performing a DDoS attack and the corresponding traffic pattern is shown in Figure 2.1b.



(a) Pulse-Wave DDoS Attack [77] (b) Traditional DDoS Attack [77]

Figure 2.1: Comparison of Pulse-Wave and traditional DDoS attack Mechanisms [77]

This pulse-wave manner of performing the attack results in a number of characteristics that make pulse-wave attacks effective at outmaneuvering DDoS defense systems:

- **High attack vector adaptivity**: Each pulse can, and typically does, make use of a different attack vector [3, 39]. Which gives it the ability to circumvent DDoS defenses that are not 'generic', meaning they are only capable of mounting a proper defense against a specific set of attack vectors [3].

- **Ability to attack multiple targets at once**: The attacking entity (*e.g.,* botnet) does not, for lack of a better term, 'cease operation' after a traffic burst, but rather changes the target with each pulse, giving it the ability to maintain an attack on multiple targets at once [16, 39, 77]. This also explains how the attack manages to achieve that rapid traffic ramp-up, as the attacking entities are not being marshalled at the beginning of the attack but rather have already been operating at the desired capacity, thus creating these immediate traffic spikes [77].

- **High longevity of attack**: Besides high attack vector adaptivity, another effect of constantly altering the attack vector when combined with the short duration of the traffic pulse is that it forces the DDoS defense to continuously rerun its analysis of what defensive measures need to be deployed to deal with the currently incoming pulse [3] appropriately. As [3] puts it, pulse-wave attacks target "the Achilles' heel of state-of-the-art DDoS defense", referring to their reaction time. This ability of pulse-wave attacks to constantly force the defense to reconsider its measures allows the attack to be maintained for long periods of time [16].

That last point in particular bears further explanation. As [77] explains, pulse-wave attacks are particularly efficient at disrupting so-called 'appliance-first hybrid mitigation' systems.

Such mitigation systems are usually sufficient to deal with the traditional DDoS attack forms which possess the already discussed slow ramp up [77]. This ramp up gives the on-premise part of the mitigation system enough time to activate the cloud-based part of the mitigation topology (hence the term 'hybrid') where incoming traffic then is being scrubbed [77]. An example of such an appliance first hybrid mitigation topology is shown in Figure 2.2.

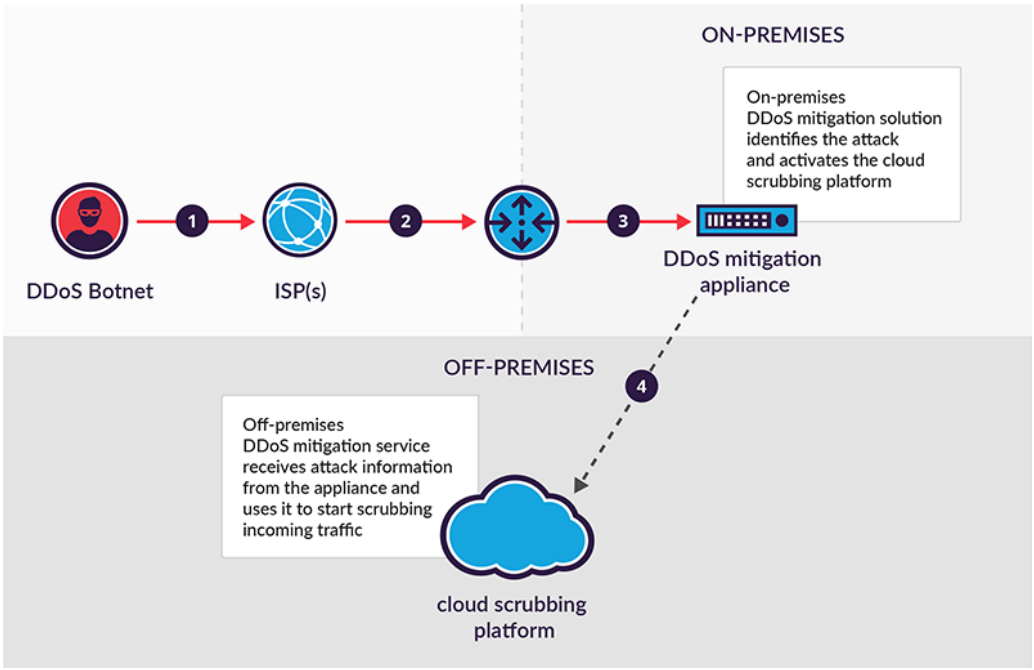


Figure 2.2: Appliance First Hybrid Mitigation [77]

With a pulse-wave attack however this does not work due to the immediacy of the attack which does not afford the on-premise component the necessary time to react and activate the cloud scrubbing, as the network is congested as soon as the attack starts [77].

Worse still, this also prevents the creation of an attack signature, meaning that even if the cloud scrubbing platform manages to get activated it does not have the necessary information about the attack to start scrubbing the traffic [77]. Instead it has to first determine the attack signature itself [77].

Such pulse-wave DDoS attacks are characterized by [39] as 'sophisticated' with a requirement of having a high degree of control over the attacking entities in order to achieve the precise orchestration which is so characteristic of pulse-wave attacks.

This leads [39] to conclude that pulse-wave attacks are likely executed using 'proprietary botnets', made up of "a relatively small number of high-capacity, connected devices (e.g., servers)."

The only statement in the literature regarding the exact attack vectors that are commonly used in pulse-wave attacks is found in [19], which specifically mentions ICMP flooding, UDP flooding, and TCP SYN flooding as possible candidates.

2.1.1.4 DDoS Attacks in Internet of Things (IoT) Networks

The topics of IoT and DDoS are intrinsically intertwined due to the role IoT devices play in events such as the series of DDoS attacks performed by the Mirai botnet in 2016 [6]. Another way in which the two topics are connected is through the attack surface IoT networks present towards DDoS attacks, resulting in attacks that exploit specific aspects of IoT networking such as those described in [1]:

- **RPL DIS Flooding attack**: A DDoS attack, where the victim node is flooded with messages, specifically RPL DIS messages used for 'neighbour discovery' (*cf.* [40]), leading to loss of node service and potentially battery exhaustion.
- **Selective forwarding attack**: This attack is characterized by selectively dropping specific packets, which can lead to Denial-of-Service when combined with *e.g.,* a 'sinkhole attack'.
- **Blackhole attack**: During such an attack, all packets are dropped, negatively affecting the IoT network topology and denying service by preventing the packets from reaching their destination.

This has lead to a collection of DDoS literature specifically focused on IoT networks, as is explored in Section 2.2.2.2

2.1.1.5 DDoS Attack Signatures

Attack signatures of DDoS attacks (also called 'fingerprints') play a key role in combating DDoS attacks and are often used by intrustion detection systems (IDS) for that purpose [20]. Such attack signatures comprise the key characteristics of observed DDoS attacks [20]. Example criteria of characteristics used in attack fingerprints include source and destination ports, average packets per second, average datarate, source IP addresses, attack duration, time-to-live values on packets and autonomous system (AS) numbers [34].

IDSs that rely on the use of attack signatures, analyze incoming traffic and compare them against their database of known fingerprints [34]. This enables the IDSs to detect if an attack is being conducted [34]. Note though, that this only works for known DDoS attacks and if attack traffic that does not match known fingerprints is analyzed by the IDSs they will not be able to recognized them attacks, at least not based on the fingerprint analysis [34]. To complement fingerprint based attack detection IDSs may use anomaly detection such that even attacks whose signatures are not known have a chance of being detected [20]

2.1.2 DDoS Datasets

Datasets containing network traces of DDoS attacks are a valuable resource for the development of defensive mechanisms such as an IDS, specifically 'anomaly-based IDSs', where machine learning (ML) models trained on those datasets are used to differentiate regular from malicious traffic [1, 5].

Given the critical role of DDoS datasets in the training process of such models and the fact that the methods and mechanisms of DDoS attacks keep evolving, having high-quality and up-to-date datasets available that appropriately capture the characteristics of attacks is key to the process of developing good anomaly-based IDSs [1, 67].

2.1.2.1 Availability of High Quality Datasets

As [67] point out in their analysis of datasets, both from real DDoS attacks, as well as gathered from simulations, none of the datasets they analyzed were up-to-date enough to include more modern DDoS attacks. Other shortcomings include insufficient real-world closeness (*e.g.,* due to insufficiently sophisticated simulation setups), anonymization of traffic data, or lack of traffic completeness [67]. Similar findings were reported by [1] and [5] in their analysis of existing datasets.

Suitable datasets from real DDoS attacks in particular are difficult to come by, as they tend to suffer from anonymization or simply being outdated, as already established. Additionally, getting access to them can come with its own set of hurdles and restrictions regarding how the data is allowed to be used, *cf.* the request form for the CAIDA 'DDoS Attack 2007' dataset [14].

In sum, this indicates that there is a lack of availability when it comes to suitable DDoS datasets or, as [67] puts it, "having a suitable dataset is a significant challenge itself".

Moreover, to the best of the author's knowledge, there are no publicly available DDoS datasets.

2.1.2.2 The Role of DDoS Dataset Generation

As established in Section 2.1.2.1, high-quality datasets can be difficult to come by, especially when the focus is put on more recent trends in DDoS attack methods. This highlights the importance of having some means to generate suitable datasets to *e.g.,* develop or refine IDSs [5].

As [67] mention in their analysis of existing datasets, simulated datasets risk being too far removed from real-world outcomes. However, generating datasets by conducting real-world DDoS attacks is both difficult and costly, thus not a viable option for most research projects [5].

This means that a setup for mimicking DDoS attacks is the prevalent method within the research for generating datasets [5]. To avoid a situation where datasets are being produced that ultimately lack real-world closeness, the configuration and the design of the generation setup are critical [5]. For example, the capabilities of the underlying tools used to implement the generator and ultimately generate traffic must be well suited to the generator's attack scenarios to mimic [5].

2.1.2.3 Dataset Generation Techniques

There exist different approaches to mimic DDoS attack traffic and ultimately generate a dataset, which all present distinct advantages and disadvantages when compared with one another [10]:

- **Real-world Traffic**: This approach represents the most suitable one when it comes to real-world closeness, as the network in which the traffic for the attack scenario is generated is constructed directly in hardware, *i.e.,* physical routers, computers and switches. This comes at the expense of flexibility, as each scenario has to be physically configured. Additionally, the used hardware is associated with significant costs and the generator cannot be shared, for when *e.g.,* the setup is to be reproduced elsewhere as part of another work or for reproduction.

- **Simulated Traffic**: Simulation represents the other end of the spectrum when compared to the real-world traffic approach. The network topology is modeled virtually and so is all the traffic generated within, and as such, there are shortcomings related to the realism of the generated traffic. However, due to its purely virtual nature, the setup is flexible, can be shared, and is low cost. Further, the approach benefits from scalability.

- **Emulated Traffic**: Emulation is a combination of aspects of the simulation and real-world traffic approaches. The network topology is simulated, but the attacker and target nodes are physical machines. As such, emulation inherits benefits and drawbacks of both the simulation and real-world traffic generation approaches, resulting in a setup that is configurable, but does incur some hardware costs, and is considered not scalable.

2.2 Related Work

The Section of the thesis is divided into three parts, the first of which serves the exploration of prior work that directly deals with pulse-wave attacks, with the second part being reserved for the examination of existing literature related to DDoS dataset generation. In the final part, the findings are discussed and this work is contrasted with existing research.

2.2.1 Pulse-Wave DDoS Attacks

Pulse-wave DDoS attacks have been given some attention in recent literature, with [9, 25, 41, 42] all recognizing them as a new form of DDoS attack.

A number of research projects have been conducted regarding **defensive measures** that are capable of recognizing pulse-wave attacks. They are explored in the following paragraphs.

[16] developed an algorithm in the form of a binary classifier that is capable of detecting malicious traffic with high accuracy. The appearance of a high numbers of unique IP addresses serves as trigger for the algorithm, signaling the start of an incoming pulse-wave attack [16]. The classifier then flags the addresses as either bots or legitimate users based on the trained classifier [16]. Unfortunately, the classifier shows a false positive rate of approximately 10 - 15%, which would lead to a significant portion of legitimate user traffic also being blocked [16]. This is solved by not blocking the IP addresses flagged as bots, but by moving them to a separate message queue with limited bandwidth, thus reducing the impact of the attack [16].

Though through a different approach, [3] also propose a mechanism to defend against pulse-wave attacks specifically. The mechanism they develop is an evolution of 'Aggregate-based Congestion Control' (ACC), a technique for dealing with high-bandwith traffic aggregates based on IP-prefixes [3]. ACC is capable of providing protection against traditional DDoS attacks, but cannot cope with the nature of traffic generated by pulse-wave attacks, as the reaction time is too slow due to so-called 'offline inference' and the configuration of the defense activation threshold introduces a tradeoff of further degrading response time or compromising accuracy through high false positive rates [3]. [3] remedy this in 'ACC-Turbo', where the inference is brought 'online' (*i.e.,* runs in the data plane), the inference mechanism is modified, and programmable scheduling is introduced. This results in an approach that is capable of providing good protection against a range of simulated DDoS attack types, including pulse-wave [3].

A number of other projects centered around defense against DDoS attacks, in general, have also shown capabilities of dealing with pulse-wave attacks or are expected to show effectiveness against them, though these works are not specifically concerned with pulse-wave attacks. [15] develop a collaborative real-time defense framework for deployment in 5G networks that has been able to provide protection against simulated pulse-wave attacks. [63] propose an 'entropy based' approach to detecting pulse-wave attacks and other difficult to deal with attack variants. Finally, related to internet exchange points (IXP) [75] introduce 'IXP scrubber', a ML-based, continuously learning system that is deployed at IXPs. Its purpose is to detect DDoS traffic, and the authors state that it should be capable of detecting pulse-wave attacks [75].

To the best of the author's knowledge, no work has been done on generating DDoS datasets that specifically deal with pulse-wave attacks.

2.2.2 DDoS Dataset Generation

Existing work done on dataset generation can be examined through different lenses. The first criterion to examine, discussed in Section 2.2.2.1, is the generation mechanism used to create the dataset, resulting in a categorization along the mechanisms discussed in Section 2.1.2.3.

The second criterion is about specialization, *i.e.,* whether the purpose of the generation is to create a specialized dataset that is more applicable to networking environments that differ from the norm in certain aspects, such as topologies or protocols. Such environments may present a different attack surface when it comes to DDoS, thus benefit from datasets that take these differences into account [1]. Alternatively, the specialization could be regarding the type of attack, by being focused on generating data for a specific DDoS attack or attack mechanism. This criterion is discussed in Section 2.2.2.2.

2.2.2.1 Examination of Existing Work regarding Dataset Generation Mechanism

Traffic generation and, thus ultimately, the generation of datasets through **simulation** is widespread in the existing literature (*cf.* works by [4, 5, 21, 43, 48, 71, 72]).

[5] for example, develop a traffic generator using the OMNeT++ simulation framework [59] to simulate a set of DDoS attack types as they might occur from different data centers around the globe. Specifically, the focus was put on UDP flooding, TCP-SYN, HTTP-GET, and ICMP flooding attacks, with the generator being configurable regarding parameters such as message size and send-intervals [5]. [5] see the generator as a promising first step in their plans to use the generated data for the development of a new IDS.

In [48], IXIA's PerfectStorm is used to simulate both normal and attack traffic against two groups of hosts, one group each for normal and attack traffic, respectively. Traffic is captured using tcpdump, allowing for further processing with the goal of the resulting dataset being the development of IDSs [48]. The generator interestingly does not only

support the simulation of Denial-of-Service attacks but is also capable of simulating different attack traffic such as backdoor attacks or worms, though the authors, unfortunately, do not provide a concrete list of specifically supported attacks [48].

[43] take a different approach by using Node-red, a middleware for connecting physical IoT devices with backend infrastructure, and write code that mimics IoT sensor output, thus simulating a set of IoT services for different scenarios such as smart home and weather stations. These simulated services, combined with a set of both attacking and normal virtual machines and a firewall, form the testbed environment [43]. This enables the generator to simulate normal traffic and attack traffic for different IoT scenarios and attacks, including DDoS attacks, specifically TCP-SYN and UDP flooding using Hping3 and HTTP-based attacks using Golden-eye [43]. [43] further train ML models on the generated datasets showing high accuracy in detecting the simulated DDoS attack types.

Emulation to generate datasets has appeared less frequently in the reviewed literature than the simulation or real-world traffic generation alternatives.

[1] propose a framework for real-time dataset generation, using Cooja and intricate network topology to emulate an IoT networking environment. The topology comprises groups of homogeneous sensor nodes and network sniffers for monitoring and data collection [1]. To further enhance real-world closeness, [1] add two 'distributor nodes' dedicated to emitting IoT-typical noise signals. The framework supports four modes of operation: normal behaviour, representing regular network traffic, and three different attack scenarios, namely: flooding using RPL DIS messages, selective forwarding and blackhole attacks, detailed in Section 2.1.1.4. Data collection makes use of Sensniff and the Libpcap library to enable the capturing of data at the MAC layer [1].

Another example of emulation is found in [10] that developed 'Botloader', a traffic generator for DDoS attacks and flash events (high load but legitimate traffic). The Botloader framework implements two distinct networks, the so called 'attack-network' and the 'management network' [10]. The attack-network is comprised of a set of host-machines that use IP-aliasing to each act as a group of attackers or legitimate users, based on the executed scenario [10]. One of the host-machines in the attack-network is dedicated to performing orchestration tasks, such as *e.g.,* configuring the scenarios executed in the other host-machines [10]. The behaviours of the machines, both for flash events and attacks, are implemented in so-called 'modules' that are applied to a host-machine during orchestration [10]. The behaviour is further configurable through a suite of parameters such as traffic rates, duration of emulation, and the number of bots [10]. The traffic emitted by the host machines is passed through a layer 3 switch, which then connects it to the target host [10]. One of the framework's shortcomings is found in the data capturing process, which in future work is to be moved to a separate machine, which would eliminate the need to run certain scenarios twice, once for capturing and once without, so the resource consumption for capturing can be accounted for [10].

Dataset generation through **real-world** traffic sees widespread use as well (*cf.* works by [2, 18, 66, 67, 72]), somewhat surprisingly so, given its cost and inflexibility, as discussed in Section 2.1.2.3.

[66] presents an architecture consisting of two separate real-world networks, the attack-network, and the victim-network. The attack network comprises a router, a switch, and four computers, running a mix of Linux Kali and Windows operation systems [66]. The victim-network is larger, consisting of three servers, ten computers, two switches, a firewall, a router, and a domain controller. One main switch port is mirrored to capture all network traffic on a separate server [66]. The computers in the victim-network run a combination of the three major operating systems by Linux, Windows, and MacOS [66]. Benign activity in the victim-network is provided by Java-based agents, developed with insights from profiling behaviours of users based on a set of network protocols, such as HTTPS and SSH [66]. The dataset generator supports a range of different attacks, some of which are DoS and DDoS attacks, implemented using tools such as Slowloris, GoldenEye, and Low Orbit Ion Canon (LOIC) [66].

Some of the authors of [66] use a similar setup in [67], where they alter the scope of the generator by focusing specifically on generating data for a diverse set of up-to-date DDoS attacks. The architecture is slightly changed by reducing the size of the victim-network and having the attack-network be provided by a third party, which is also responsible for conducting the DDoS attacks [67]. This generator supports twelve different DDoS attack profiles, including WebDDos, LDAP and SYN attack scenarios [67].

Another example of real-world traffic-based dataset generation can be found in [18], where the network comprises parts of a real-world in-use academic network, spanning the five largest cities in Lithuania. This allows for capturing actual real-world normal behaviour traffic, as in real-world traffic that does not stem from artificially generated traffic in a sandbox environment [18]. The generator supports twelve network-attack types, including DoS and DDoS attacks, such as TCP-SYN, HTTP-flooding and UDP-flooding [18].

If the term 'generation' is applied less strictly, then there is also the option to create datasets without actually generating any traffic, real or simulated, which in this thesis makes the term '**transformation**' more suitable. An example of dataset generation through transformation can be found in [31], where benign datasets are injected with attack-activity, thus synthetically generating a new dataset that contains DDoS attack traces.

2.2.2.2 Literature Focused on Generation of Datasets for Specialization

The topics of DDoS and IoT are related, as hinted at in Section 2.1.1.4. Unsurprisingly, the literature review has identified a number of works that focus on generating DDoS datasets that satisfy specific qualities of IoT networks (*cf.* works in [1, 21, 43, 72, 73]. For example, [1] present a generation framework, which takes into account IoT-typical levels of signal noise and attacks that exploit topologies and protocols used in IoT networks. [43] develop a testbed environment that is capable of simulating IoT scenarios such as smart homes and weather stations and then incorporate these into their attack scenarios. As a final example, [73] propose a new dataset for IDS development derived from a simulated environment that refines existing work, such as the aforementioned [43], by adding more regular traffic features and optimizing other parameters.

Another area of specialization is termed 'vehicular ad hoc networks' (**VANET**s), with works such as [4]. [4] present a dataset that takes into account VANET characteristics,

such as rapid mobility of network nodes leading to topology changes and speed and density of vehicles leading to variable connectivity. They simulate a realistic highway scenario for vehicle-to-infrastructure DDoS attacks [4]

Container-based environments, like those found with **Docker**, represent the last area of specialization encountered in the literature review. [71] in particular provide an example of work done in this area, by proposing the first dataset that takes into account and is generated within a containerized generator implemented using Docker and Docker swarm.

It should be understood that this list of areas of specialization should not be seen as complete, as those are simply the areas encountered during the literature review, which is not guaranteed to be exhaustive.

2.2.3 Discussion

The examination of existing work done concerning pulse-wave attacks and DDoS dataset generation has yielded a few key insights.

First of all, whilst pulse-wave attacks are mentioned in the literature regarding emerging threats in the domain of cyber-security in as early as 2017 (in [9]), they still have not seen much attention when it comes to developing defensive measures, with only a small number of projects being dedicated to dealing with pulse-wave attacks in detail. Further, to the best of the author's knowledge, no work has been done on developing a dataset or a dataset generator for pulse-wave attacks.

Moreover, the examination of work that is specifically dedicated to building a dataset generator or that builds a dataset generator as part of the project shows that the methods of simulation and real-world traffic generation are more popular than emulation. Whilst the prevalence of the simulation approach is to be expected, given its flexibility and low barrier of entry, it is surprising to see that more projects have chosen real-world traffic generation over emulation, since real-world traffic generation typically is associated with certain hardware requirements (and costs) and suffers from inflexibility and an inability to share or reproduce the setup (*cf.* Section 2.1.2.3). Lastly, to the best of the author's knowledge, all of the examined works dealing with dataset generation have focused solely on the victim's perspective, *i.e.,* the traffic and impact as seen on the target machines or network.

Table 2.1: Related Work Overview

Work	**Purpose**	**Specialization on Specific DDoS Type**	**Distributed Perspective**
[5]	Traffic Simulation	✗	✗
[4]	Traffic Simulation	VANET	✗
[71]	Traffic Simulation	Docker	✗
[43]	Traffic Simulation	IoT	✗
[48]	Traffic Simulation	✗	✗
[21]	Traffic Simulation	IoT	✗
[73]	Traffic Simulation	IoT	✗
[1]	Traffic Emulation	IoT	✗
[10]	Traffic Emulation	✗	✗
[66]	Real-World Traffic	✗	✗
[67]	Real-World Traffic	✗	✗
[72]	Real-World Traffic	IoT	✗
[2]	Real-World Traffic	✗	✗
[18]	Real-World Traffic	✗	✗
[31]	Dataset Transformation	✗	✗
[3]	Attack Mitigation	Pulse-Wave	✗
[16]	Attack Mitigation	Pulse-Wave	✗
This work	Traffic Simulation	Pulse-Wave	✓

Table 2.1 presents an overview of the examined literature, showing that this work presents a novel approach by combining a focus on simulating pulse-wave attacks with a distributed perspective, *i.e.,* the implementation of a distributed pulse-wave DDoS dataset generator that aims to present data about the captured traffic at different points in the network on the path from sender to recipient.

Chapter 3

Design

In this chapter, the requirements for the system-to-be are explored, and the design of the prototype is captured. Section 3.1 lists the system requirements. The primary use case implemented by the prototype is described in detail in Section 3.2. Finally, the prototype's architecture is discussed in Section 3.3.

3.1 Prototype Requirements

The following requirements were formulated for the prototype system:

- **R1: Distributed Perspective**: The prototype shall be able to capture traffic at different points of the modeled topology for later analysis to have more data to work with than with a single point of view, resulting in a more complete view onto the attack.
- **R2: Variability of Pulse-Wave Patterns**: The prototype shall be flexible regarding what kind of pulse-wave attack patterns can be produced, regarding factors such as the attack vectors, the duration, and the amount of traffic of a given pulse.
- **R3: High Degree of Configurability**: The prototype shall be built such that configuration is possible for users not adept at writing code, *i.e.,* configuration shall be possible without making changes to the prototype's code base. Furthermore, the configuration options shall be understandable and, where possible, sensible default settings shall be used to reduce the burden of configuration on the user.
- **R4: Reproducability of Results**: The prototype shall be able to consistently reproduce comparable results, given that it is ran under comparable circumstances and with identical configuration, thus allowing for both the verification of results and the reuse of results by other users for their work. For that purpose, the prototype shall also be publicly available and have clear instructions regarding installation.
- **R5: Extendability**: Given the prototype nature of the system-to-be, the system shall support extension through *e.g.,* introducing additional attack vectors or different topologies.

3.2 Application Scenario

The prototype system implements a *generic primary scenario*. *Generic* because the scenario's goal is to serve as a basis for crafting *specific use cases* through the application of configurations.

Primary because while the prototype only explicitly implements this particular scenario, the prototype's components are also built to be reused to construct, or contribute to, one's own separate use case. Note, though, that this does require interacting with the code directly and cannot be done just through configuration. Refer to the Implementation part of the thesis (Chapter 4) for more information on the how the components are implemented.

As outlined in the introductory chapter (*cf.* Section 1.1), the distributed perspective is a key aspect of the system-to-be, with different entities collaborating and pooling their captured traffic traces to establish a more holistic view of an attack.

A good example for such an entity is an IXP. IXPs allow their connected members (typically each an Autonomous System (AS) [68]) to exchange traffic between each other through the infrastructure of the IXP, as an alternative to sending traffic via the global internet [70].

3.2.1 IXP and AS Topology

This relationship between IXPs and ASs is what the primary scenario is based on. Figure 3.1 illustrates the kind of topology that results from this relationship, with IXP nodes forming connections amongst each other, allowing their connected ASs to communicate with other ASs that are also part of this topology.

The number of IXP nodes and the number of ASs connected to each IXP node are subject to configuration. The semantic meaning of the individual IXP nodes is left to be determined by whatever scenario the user wants to configure. For example, the IXP nodes could all be part of the same (IX) Internet Exchange, such as *e.g.,* the SwissIX ([69]), or they could all be part of different but collaborating IXs. The IXP nodes also represent the points in the topology where traffic traces are captured.

3.2.2 AS Internals

Figure 3.1 also provides a close-up view of one of the examples ASs. The prototype implements three main types of components that populate a given AS; client nodes, server nodes, and an AS gateway.

The AS gateway serves as the entry and exit point to and from the AS. In other words, all traffic that crosses the threshold of the AS in either direction traverses through the AS gateway. The AS gateway is also what connects a given AS to the IXP node it belongs to.

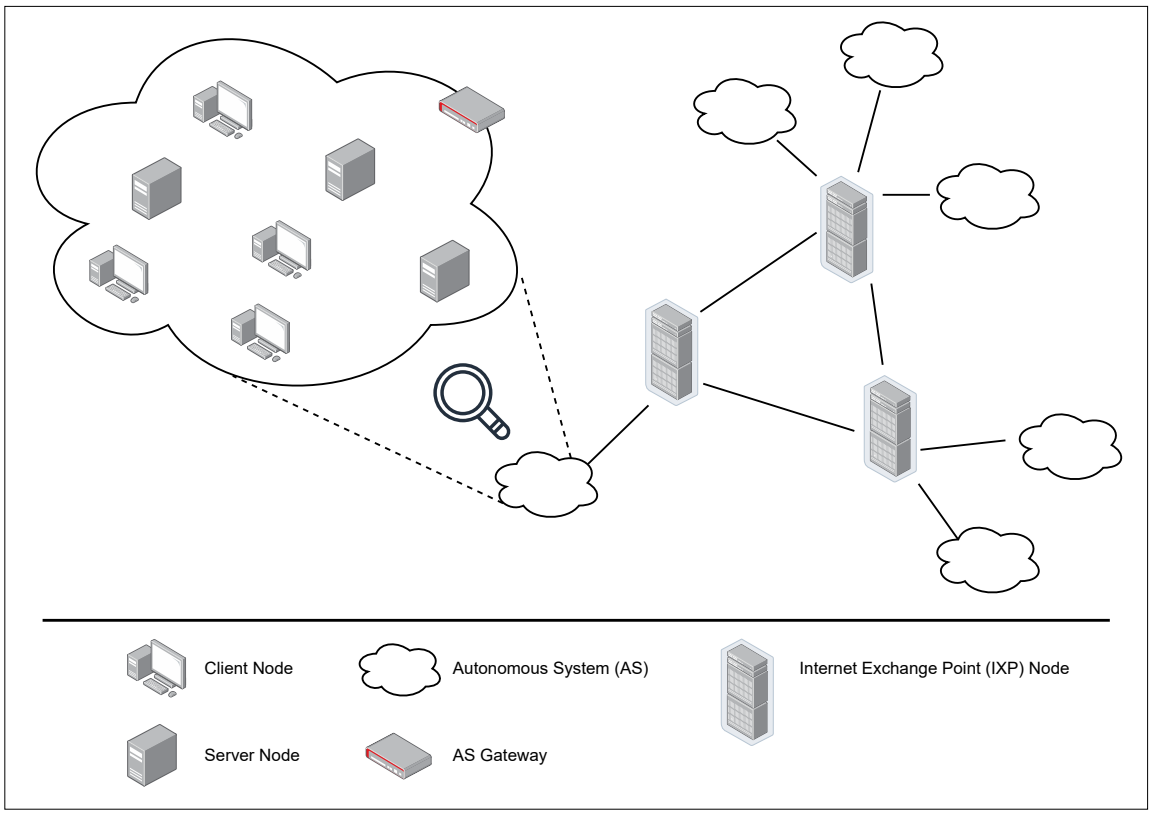


Figure 3.1: Primary Scenario Example Topology

Client nodes are nodes that perform some action with a server node. Benign client nodes represent legitimate users of a given server, making use of whatever service the server offers. Malicious client nodes (*i.e.,* attackers) perform coordinated pulse-wave DDoS attacks against a subset of the server nodes. The number of client and server nodes within each AS are subject to configuration and the same applies to which subset of server nodes is part of the list of targets that are attacked by malicious client nodes.

The interactions between client and server nodes are not restricted to a given AS. Instead, any client node has the capability to interact with any server node in any of the ASs that are part of the topology. In fact, given that traffic is captured at the IXP nodes, any intra-AS traffic never passes through any IXP node and is therefore not represented in the captured traffic traces.

3.2.3 Attack Configurability

The attacks performed by malicious client nodes can also be finely tuned through configuration regarding aspects such as the types of attack vectors that are being used, the duration of the pulses, the time it takes to switch the attack from one target to the next, the attack data-rate coming from each malicious client as well as the size of the packets, and the source and destination ports.

The purpose of this configurability is to allow for the creation of use cases that result

in distributed datasets that differ regarding properties which are typically used for attack signature creation (*cf.* Section 2.1.1.5). The exact set of characteristics that will be configurable either directly or indirectly depends on what is feasible within the traffic simulation framework that is ultimately chosen and the timeframe of the thesis.

Desirable characteristics include the protocol used in an attack vector, attack duration, average data rate and packet volume, source and destination ports, IP addresses and the AS number [34].

3.2.4 From Generic Scenario to Specific Use Case

All this combined allows the prototype to execute a wide range of diverse *use cases* through the application of configuration to the primary scenario.

If the user wants only to capture attack traffic traces of one specific attack vector but with different data rates or packet sizes, then that can be configured. If the user wants to have all attackers in one AS and all targets in another AS, then that can be configured. If the user wants to have only one attack target, but have it hit with a variety of differently configured attack vector pulses, then that can be configured too.

Ultimately the purpose of this design is about providing **flexibility** in terms of what use cases can be simulated and thereby also ensuring that a wide range of distributed pulse-wave datasets.

3.3 Architecture

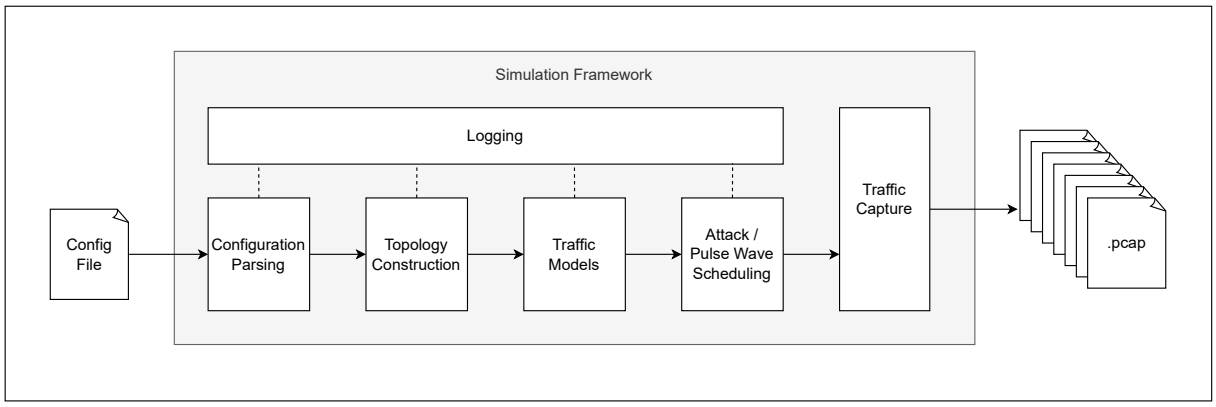


Figure 3.2: System Architecture

Figure 3.2 depicts the non-technology-specific architecture of the system and shows the main system tasks in the form of components. Additionally, the arrows indicate the flow from configuration input to PCAP output.

3.3.1 Configuration Parsing

Configuration parsing encapsulates two key activities:

- **Read from file:** The first part of configuration parsing consists of identifying the configuration file and reading it into the system for further processing. This includes running some basic validation on key parts of the input.
- **Fill missing optional values:** The second part of configuration parsing is completing the configuration input, *i.e.,* filling in default values for optional parameters that have not been specified in the configuration file.

An alternative approach to dealing with default values is to have them already be present on whatever class or object will end up implementing the desired feature, instead of having the default values be part of the configuration parsing.

The reason for having the default values be filled in during configuration parsing is that it enables providing logging output of the entire configuration, including default values, all at once. This provides the user with a more complete picture of what the final configuration of the system actually looks like, once the properties that were not explicitly specified in the configuration file are added. This would be more difficult to achieve, if default values were distributed all throughout the system.

3.3.2 Topology Construction

The next task of the system is to construct the topology resulting from the parsed configuration. Specifically, this includes the following steps:

- **Determining central network (CN) topology**: Building up the CN, *i.e.,* the network of IXP-nodes with the configured IXP-nodes and with the correct settings regarding aspects such as bandwidth or delay.
- **Instantiating ASs and connections to CN**: Creating the ASs according to the configuration and connecting them to the CN topology at the correct IXP-node.
- **Creating nodes within each AS**: Creating the configured amount of each type of node within each AS.

3.3.3 Traffic Models

Once the topology has been established, each type of node within an AS needs to be able to make use of the appropriate traffic models. For example, benign nodes need to have access to a traffic model that allows them to generate background traffic when connecting to a server node. Similarly, attacker nodes need to have access to traffic models that allow them to create attack traffic that corresponds to the configured DDoS attack vectors.

3.3.4 Attack- & Pulse-Wave Scheduling

Given the nature of pulse-wave attack, the attack traffic needs to follow a schedule that determines when which attack vector is active and which target is currently being subjected to the DDoS traffic.

This schedule is shared amongst all attacker nodes, due to the attack being executed in a coordinated manner, thus it needs to be made available to all attacker nodes that are part of the topology.

3.3.5 Traffic Capture

The final main component in the system is tasked with capturing traffic at the correct locations of the topology and providing them to the user by producing PCAP files on the local file system.

3.3.6 Logging

The system benefits from logging capabilities as they can help making the system easier to understand. This is true, particularly during development, but also for the end-user, as having some information regarding what the system is currently doing can be helpful. In that sense, logging is not strictly required to build a functioning system but is still desirable.

In this case, logging is primarily an important part of the configuration, as it provides the user with insight about the final configuration, once default values have been used to fill gaps in the configuration file (*cf.* Section 3.3.1). In fact, all parts of the system can improve the user-experience by making use of logs, with the exception of the traffic capture component that already produces user-oriented output in the form of PCAP files.

Chapter 4

Implementation

In this chapter, the details of the system implementation are discussed. To begin, the selection of the network traffic simulation framework is justified in Section 4.1. Afterwards, the high-level structure of the prototype as an NS-3 framework is outlined in Section 4.2. Finally, the implementation of the components defined in Section 3.3 is outlined, discussing each of them in turn.

Section 4.3 outlines the inner workings of the 'main' script which acts as the coordinating entity of the system. Configuration parsing and validation is discussed in Section 4.3.2 and the construction of the simulation topology is elaborated in Section 4.3.3. This is followed by an explanation of the traffic models (Section 4.3.4) and attack scheduling processes (Section 4.3.5). Finally, Section 4.3.6 provides insight into the traffic capture process.

Note that as per agreement with the supervisor of this thesis, AI tools such as ChatGPT [58] have been used for the purpose of coding assistance. However, none of the work presented in this report and none of the code has been produced by AI, as the tools were only used for ideation processes.

4.1 Framework Selection

The decision was made to implement the prototype with the NS-3 discrete event simulator. The reasons for that are as follows:

- **Well established**: NS-3 is a common choice within academia regarding network simulation. A brief serach for "ns-3 network simulator" on Google Scholar [30], reveals dozens of hundreds of both recent and older publications.
- **Actively maintained**: NS-3 has had multiple releases this year already, with the latest (v3.39 [56]) having occurred less than a week ago at the time of writing this part of the report.

- **Strong library support**: NS-3 boasts an extensive list of library modules that cover a wide range of topics and features ranging from different routing models to parallelization using the message passing interface (MPI) standard [57].
- **Open for extension**: NS-3 is designed to support code contributions in the form of modules, with the documentation providing detailed instructions on how to set up and develop a new module [54].
- **Built-in logging and tracing**: NS-3 comes with built-in logging capabilities and has a rich tracing system that, among other things, enables the capturing of traffic in PCAP files [54].
- **Extensive documentation and example scripts**: NS-3 is well documented regarding tutorials, code examples, and code API [54].
- **Strong community support**: NS-3 has a number of highly active community members that offer their advice on NS-3 related issues on an open Google group forum [51].
- **Previous experience within department**: Other members of the department, specifically within the Communication Systems Group (CSG) ([74]), have had experience with NS-3 or are currently working on other projects that utilize NS-3 , thus enabling the exchange of ideas, insights, and learnings.

This is not to say that other candidates, such as *e.g.,* OMNet++ [59], also a discrete event simulator, do not have comparable qualities and are not also valid choices, but as a whole NS-3 made the most sense for this thesis, especially when taking into account other members of the department already having had hands-on experience with the framework.

4.2 Implementation as NS-3 Module

As mentioned in Section 4.1, one of the reasons for choosing NS-3 is its design that allows for the creation of modules that represent an easy way to integrate external code into an NS-3 installation. This prototype is consequently implemented as such an NS-3 module.

The documentation contains detailed instructions regarding how such a new module is to be constructed ([54]), thus, this section is limited to discussing the folder- and file-structure created within the new module.

Listing 4.1: Top-level structure of new NS-3 Module

```
distributed-pulse-wave-simulator
    ↪ /external
    ↪ /helper
    ↪ /model
    ↪ /service
    ↪ /wrapper
    ↪ CMakeLists.txt
```

Listing 4.1 shows the top-level structure within the created module. The role of the CMake file `CMakeLists.txt` is discussed at length in the documentation, but to still briefly explain its purpose on a high level: it is responsible for defining which files make up the new module, *i.e.,* are to be respected during the build process.

The directory `/external` is reserved for external libraries that are not NS-3 related but that has been included within the module to make the installation process simpler by not having to download files from other sources. It only contains a single file, `rapidyaml-0.5.0.hpp`, which is the all-in-one header-only version of *rapidyaml* ([27]), a library that provides `.yaml` file parsing capabilities. Its used during the configuration parsing is discussed in more detail in Section 4.3.2.1.

The directories `/helper` and `/model` contain files that operate on the level of NS-3 helper- respectively, model-implementations. In other words, any files that follow the implementation structure of NS-3 , thus *e.g.,* make use of the built-in `TypeId` system, are contained in these two directories. This includes both files by other authors, such as the HTTP traffic model by [28], and modified versions of existing NS-3 files that don't rely on being placed directly into the NS-3 installation's `src` folder.

Meanwhile, the `/wrapper` directory contains the files that don't implement completely new models or helpers but instead make use of the NS-3 implementation and ways that ultimately make up the majority of the prototypes logic. This includes the implementations of the AS, the CN and the individual node types (benign, malicious, target, non-target). The 'wrapper' name aims to hint at that fact, given that the files contained within 'wrap' around functionalities provided by NS-3 , instead of directly extending existing models.

Finally, the `/service` directory contains a number of helper classes that provide functionalities that are not part of any specific wrapper file but have to be available more globally. This includes, for example, a helper class responsible for computing the attack schedules based on the number of targets, the number of attack vectors, and their respective configurations. Having that functionality available in a separate helper class means not having to re-compute the entire schedule for each attacker node.

4.3 Component Implementations

In this section, the components outlined within the architecture diagram (Figure 3.2) are revisited and brought into context of the NS-3 framework. Specifically, it outlines how they are implemented in class hierarchies, what implementation decisions were made and how the individual components interface with each other.

Where appropriate, code excerpts are discussed in detail, though it is to be understood that this section does not aim to be full-fledged documentation. If one is interested in inspecting the code documentation, one may visit the GitHub page ([61]) where the source code is made available to the public.

It also needs to be established that the term 'node' in the context of the implementation is **ambiguous**, as NS-3 has its `Node` type and class hierarchy, whilst this system has its

own set of node classes that relate to node roles and behaviors (malicious, benign, etc.). To distinguish them clearly, when relevant, the term 'DPWSNode' is used to refer to this system's node class hierarchy and 'NS3Node' to refer to the NS-3 classes. If neither term is used the reader shall assume that the distinction is not relevant to what is being discussed and the two terms can then be seen as synonymous.

4.3.1 Main Script

One component that was not part of the aforementioned architecture diagram is the 'main script'. The main script contains the `main` function that is being executed by the user when giving the run command through the command-line interface (CLI). The main script is responsible for a number of activities, specifically:

- **Parsing CLI arguments**: When an NS-3 script is run, the user may choose to supply a number of predefined arguments alongside the run command, such as *e.g.,* the name of the configuration file or the control flag that determines whether the configuration should be printed to the console as a whole once it is parsed.

- **Instantiating components**: The main script is tasked with instantiating the necessary components in a way that adheres to the configuration and the necessary sequence of events, *e.g.,* instantiating the CN before starting the instantiating the ASs, as they depend on having CN nodes present to attach to.

- **Control flow**: The main script does not only instantiate components but also establishes the necessary flow of information between them by passing information such as *e.g.,* the list of IPv4 addresses of the target nodes to the attacker nodes or by providing access to helper classes to components that rely on being able to share specific instances of those helpers. Furthermore, the main script is also responsible for triggering the construction of the routing tables.

- **Anchoring parallelization**: The system can make use of MPI to parallelize specific tasks and increase overall performance. This is initiated within the main script and the main script also is responsible for assigning 'rank', *i.e.,* specifying which parallel instance is responsible for which tasks.

- **Controlling the NS-3 simulator**: The NS-3 simulator instance expects to be managed in terms of being started and shut down and can be configured in a number of ways, such, for example, setting the time resolution at which the simulator operates. This is also done in the main script.

There are two specific sections of code within the main script that warrant a more detailed discussion, namely the parallelization management and the so-called `NodeLookupMapper` helper class which is a key part of the instantiation phase.

4.3.1.1 Managing MPI

NS-3 comes with module support for MPI, which in turn gives access to the so-called `MpiInterface` that hides much of the complexity of having to work with MPI directly [55]. Enabling MPI for a specific simulation run requires passing a specific CLI argument, specifically `mpiexec -np {numcores} %s`, where 'numcores' is an integer that specifies how many logical cores the system is allowed to use [55].

For this to take effect, the rank (think of it as the 'id' of a given parallel instance) of the parallel instances must be managed in a way that satisfies the following criteria:

1. MPI can only split a topology along a `PointToPoint` channel, which results in a special 'remote' PointToPoint connection if the two nodes are not within the same rank. Only in this way can a topology be split up for parallelization, thus every node that is created has to be assigned to a specific rank. Implementing the control flow that determines which rank a given node is the responsibility of the main script.

2. Just as each node must be assigned a specific rank, the act of tracing (*i.e.,* capturing traffic in a PCAP) is specific to a given rank because packets are only 'visible' on the topology within the specific rank that manages that set of nodes [55]. Thus, care must be taken that the ranks are assigned in such a way that capturing traffic is still possible in a convenient way.

3. Finally, the parallelization has to make sense regarding the way it makes use of the granted number of cores. For example, granting two cores to a system that implements parallelization in a way that expects 3 parallel instances to each have their own core will result in erroneous behaviour.

Listing 4.2: Logic for Rank Assignment

```
1  bool mpiMultiThreading = MpiInterface::GetSize() > 1;
2  std::vector<uint32_t> rankToAsIndex;
3
4      if (mpiMultiThreading)
5      {
6          int numLogicalCores = MpiInterface::GetSize();
7          int numAS = aSConfigVector.size();
8          int totalTasks = numAS + 1;
9          if (totalTasks <= numLogicalCores)
10         {
11             for (int i = 0; i < aSConfigVector.size(); i++)
12             {
13                 rankToAsIndex.push_back(i + 1);
14             }
15         }
16         else
17         {
18             for (int i = 0; i < aSConfigVector.size(); i++)
19             {
20                 rankToAsIndex.push_back((i + 1) % numLogicalCores);
21             }
22         }
```

```
23      }
24      else
25      {
26          for (auto _ : aSConfigVector)
27          {
28              rankToAsIndex.push_back(0);
29          }
30      }
```

Listing 4.2 shows the section of code responsible for performing rank assignments, though it has been stripped of comments for the sake of brevity. The logic therein handles both the concerns of the 2nd and 3rd criterion outlined above. The 'unit of parallelization' that has been chosen is the AS. This is primarily because it is a convenient place to do so, given that its gateway has exactly one PointToPoint connection to some CN node, and all nodes within the AS then can be created with the same rank, not having to further differentiate within the AS. A further effect of that is that this makes it easy to have the CN operate within its own rank. Lastly, it allows for the AS to freely chose how it models its internal topology (*i.e.,* is not limited to PointToPoint connections, as it does not have to worry about internally splitting up into multiple ranks), which also contributes to design requirement **R5**, by making the system extendable through additional AS implementations, should the need arise.

Figure 4.1 provides an illustration of what the resulting rank assignment across the topology looks like, given the example of having three ASs, with each color (green, blue, red) representing a rank greater than zero, with rank zero being depicted in white. Having rank zero reserved for the CN solves the problem of having traffic capture happen on different ranks, given that all interfaces on which traffic is ultimately captured belong to nodes that are part of the CN, which now has its own dedicated rank.

Getting back to Listing 4.2, the `MpiInterface::GetSize()` call (line 1) provides the total number of instances. If it that number is 1, then only one logical core is available and no parallelization occurs, resulting in having rank zero assigned to every node in the entire topology (lines 24-30), by setting the rank allocated to each AS to zero.

If multiple cores are made available, then there is still a further case distinction to be made. The number of total tasks that ideally each have their own core is calculated by summing up the number of ASs and adding one for the CN (lines 7, 8). Should the total number of tasks be equal or less than the allocated number of cores, then each task can run on its own rank (lines 9-15). If that is not the case, then problems will occur.

For example, going back to Figure 4.1, the total number of tasks is 4, but assuming the user only granted 3 cores, then one of the ASs would receive a rank that does not actually coincide with a logical core, thus would not be executed. To solve this, a module calculation is used that naively distributes ranks based on the number of available cores, thus ensuring that only valid ranks are assigned (lines 16-22).

Note furthermore the `i+1` on line 20. It is there to ensure that rank 0 is the first rank to receive a second task, as it is "only" responsible for the CN, thus not tasked with any traffic generation, which the author expects to have higher performance cost. This avoids

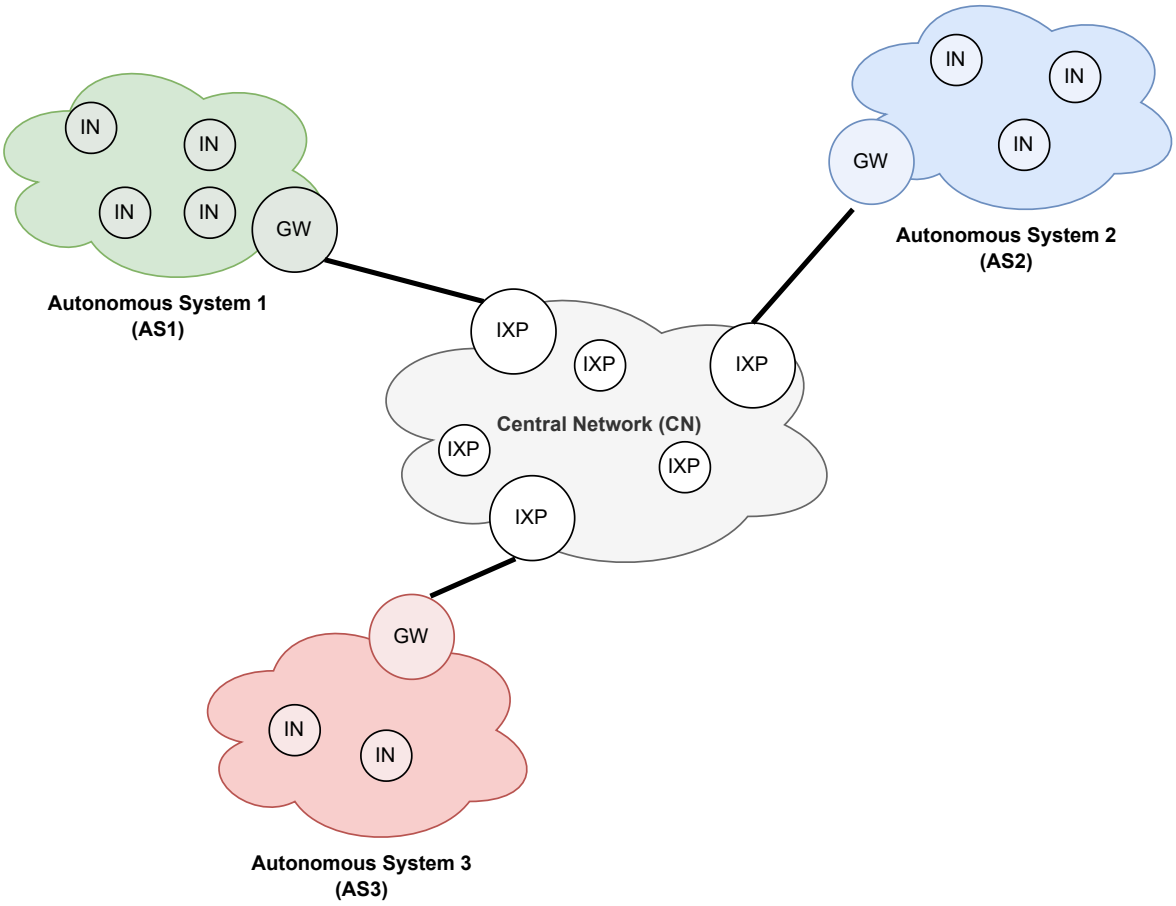


Figure 4.1: MPI Rank assignment

a situation where *e.g.,* with four total tasks and three cores the CN still has its own dedicated rank, whilst another rank has to handle two ASs.

Having now performed the rank assignment across the ASs and the CN, the 1st point still needs to be satisfied. This is achieved by simply passing the assigned rank for each AS as an argument during construction, which then enables the AS to assign the rank when it instantiates its nodes. Finally, the rank assigned to nodes within the CN will always be zero, no matter if multiple cores have been allocated or not, thus the rank of the CN nodes can simply be hard-coded.

4.3.1.2 Map for Node Lookups

It is essential for information about individual nodes to be accessible within the main script, such that *e.g.,* a benign client node can be given the IPv4 address that was assigned to its peer (some server node) or to enable the construction of the target list, by retrieving the IPv4 addresses of all target servers and providing them to the attacker nodes.

Address assignments happen during the construction of the CN respectively the individual

ASs. Thus the simplest way to get access to such addressing information is to chain the respective function calls. For example, each AS exposes functions such as `GetIpv4ByNodeId` that then internally resolve to the specified node and in this case call that nodes `GetAssignedIpv4Address` method and pass the return through to the callee.

However, all of this operates based on Node Ids, meaning the main script needs to have some way to know in which AS, which Node is present such that it can retrieve the necessary information from that node. To make that lookup process more convenient than having to construct nested loops and naively iterate through lists, the `NodeLookupMapper` class was built.

It first comes into play during the configuration parsing process, where one of its two internal maps is populated. The map contains a key-value pair that tracks the relationship between Node Ids and AS Ids, thus enabling a lookup from Node Id to AS Id or in other words determining in which AS a node with a given node id resides.

Listing 4.3 shows relevant excerpts from the main script that highlights the way the `NodeLookupMapper` operates. Lines 1 and 2 show it being passed to the configuration parsing, in which the first map is populated, as outlined above.

Listing 4.3: Use of the `NodeLookupMapper`

```
1  NodeLookupMapper nodeLookupMapper;
2  ConfigFileReader config(configFileName, &nodeLookupMapper);
3  ...
4  std::vector<P2pAutonomousSystem> aSVector;
5  int asVectorIndex = 0;
6  ...
7  for (auto asConfig : aSConfigVector)
8      {
9          // instantiate AS for asConfig
10         ...
11         // update mapper
12         nodeLookupMapper.AddAsToAsIndexEntry(asConfig.GetId(), asVectorIndex);
13         asVectorIndex += 1;
14         ...
15         aSVector.push_back(aS);
16     }
17 ...
18 for (auto bConfig : bCConfigVector)
19 {
20     // retrieve server connection information for configured peer
21     std::pair<Ipv4Address, int> peerServerInfo =
22         aSVector[nodeLookupMapper.GetAsIndexByNodeId(bConfig.GetPeer())]
23             .GetHttpConnectionInfoByNodeId(bConfig.GetPeer());
24     ...
25     // can now instantiate benign node now that peer connection info is known
26 }
```

Later, during the instantiation of the ASs, their index (line 5) in the vector that will end up holding them (line 4) is utilized in line 12 to populate the second map within the `NodeLookupMapper`.

The second map maps AS ids to the position of that AS within the `aSVector`. Both maps combined now allow for lookups such as the one shown in lines 21-23, where the connection information for the peer configured on a benign node is retrieved.

4.3.2 Configuration Parsing

Configuration parsing is handled by a set of classes within the `/service/configuration` directory of the module. Specifically, the `ConfigFileReader` is responsible for the parsing of the configuration file, whilst the class hierarchy created by the fields within `/service/configuration/objects` is instantiated in a one-to-one relation to that respective wrapper classes that will need to be instantiated in the main script.

For example, for each configured attack vector an instance of `AttackVectorConfiguration` will be created, whilst for each configured AS an instance of `AutonomousSystemConfiguration` will be constructed. Where appropriate, one configuration class instance might hold collections of other configuration classes, such as *e.g.,* in the case of the `CentralNetworkConfiguration` which holds a vector of `NodeConfiguration` that represent the nodes within the CN.

It is worth noting, that these configuration classes do not yet make use of any specific NS-3 provided types, *e.g.,* network addresses are stored as strings rather than `ns3::Address` or `ns3::Ipv4Address` and data rates are also stored as string rather than their corresponding NS-3 type `ns3::DataRateValue`. This leads to the parsed configuration being agnostic of NS-3 being used, thus allowing for maximal flexibility regarding how the parsed values are utilized, contributing to design requirement **R5**.

Each of these configuration classes inherits from a common parent class, which enforces the implementation of a `PrintConfiguration` method which then can be used to log the entire configuration to the console. More importantly, the presence of a shared parent class also allows for further functionality to be dictated top-down should the need arise in the future.

This approach of parsing the configuration file into what essentially amounts to just another packaging of said configuration may appear to be an odd choice at first, but there are reasons to do so:

- **Parsing all done at the start**: Using this approach to configuration parsing, the entire parsing process is completed before any of the wrapper classes are instantiated. This allows for validation to occur before attempting to construct class instances and further enables a simple way to log the configuration to the console, providing the user with a view of the complete configuration, including the default values the system used to fill in the gap where the configuration file does not specify values for optional configuration settings.
- **Parsing happens only once**: Parsing happens once at the beginning and the resulting configuration classes are then used for instantiation, instead of having to continuously parse specific sections of the configuration file whilst the main script

constructs the wrapper classes, having to have the file kept open all throughout or worse still, reopening and closing it constantly. Therefore this approach to parsing constitutes a more efficient way of interacting with the file system.

- **Simplified constructors**: Constructors of the wrapper classes can now rely on the fact that they receive the majority of the information they need for construction in a single argument in the form of such a configuration class. This avoids having to provide different constructor signatures for different combinations of settings being present or absent within the configuration file.
- **Wrapper classes are unaware of the parsing process**: Having default values be filled in during parsing at the start removes the need for the wrapper classes to be aware of those default values. This ultimately results in wrapper classes being entirely unaware of the entire parsing process, resulting in a cleaner separation of responsibilities.

The creation of these configuration classes is straightforward and is not discussed in this report. Rather, the focus is put onto 'rapidyaml' and how it is used within this project to perform the actual reading of the configuration file. Additionally, the structure of the configuration file is outlined. Finally, parts of the validation that is run during parsing is explained as well.

4.3.2.1 Parsing with rapidyaml

This work makes use of the 'rapidyaml' [27] library developed by GitHub user 'biojppm'. As described on the project's GitHub page, it represents a simple yet performant way of reading YAML (Yet Another Markup Language) files [27]. YAML was chosen as the configuration format for this project due to being, as stated in [22], "focused on human readability" whilst also allowing for comments, thus contributing to design requirement **R3**.

The *rapidyaml* library is available under the MIT-License and can be included in the project as a single header file. Once the library is set up within the prototype project and the functions for interacting with the file system (as shown in rapidyaml's quickstart guide on its project repository) are in place, it can be used as shown in Listing 4.4, which displays excerpts of the function `ConfigFileReader::ParseCentralNetworkConfiguration`.

Listing 4.4: Using 'rapidyaml' for Parsing

```
1  ryml::ConstNodeRef root = m_rawTree;
2  if (!root.has_child("central_network"))
3  {
4      NS_FATAL_ERROR("Configuration does not contain any 'central_network' key.");
5      return;
6  }
7  root = root["central_network"];
8
9  std::vector<NodeConfiguration> centralNetworkNodes;
10 for (ryml::ConstNodeRef node : root["nodes"].children())
```

```
11 {
12     if (node.has_child("id"))
13     {
14         NodeConfiguration n(ToString(node["id"].val()));
15         centralNetworkNodes.push_back(n);
16         ...
17     }
18     ...
19     // parse optional settings for each CN node
20 }
```

Line 1 shows the type provided by *rapidyaml* that is used to hold the parsed 'tree' resulting from reading the configuration file. That `ConstNodeRef` type then is also used to traverse the parsed tree, *i.e.,* jump to a specific node as in line 7 or iterate through a list of child-nodes as in line 10.

On each `ConstNodeRef` methods are available to check for specific child nodes (line 2), access all child nodes (line 10), or access the value of a specific node (line 14). The `ToString` method shown in line 14 is used to convert the result of reading the value at a specific node from rapidyaml's `csubstr` type to the standard library's `string` type.

This taken together is essentially all that is required to traverse and parse the entire configuration file.

4.3.2.2 Configuration File Structure and Options

Having established how the system parses YAML files, a look at how the configuration file is structured and what options are configurable is warranted. The file contains the following keys:

- `global_settings`: Specifies global settings
- `central_network`: Contains CN configuration
- `autonomous_systems`: AS configurations
- `attacker_nodes`: Attacker node specifications
- `benign_client_nodes`: Benign node specifications
- `target_server_nodes`: Target server node settings
- `non_target_server_nodes`: Non target server node configurations

Each of these keys expects a number of required and accepts a number of optional properties for which default values are present in the configuration classes if they are omitted in the configuration file.

Listing 4.5: Global Settings Configuration

```
1  global_settings:
2      capture:
3          pcap_prefix: test_scenario
4      attack:
5          burst_duration_s: 25.0
6          target_switch_duration_s: 0.0
7          attack_vectors:
8            - type: icmp_flooding
9            - type: udp_flooding
10             packet_size: 256
11             data_rate: 1Mbps
12             burst_duration_s: 45.0
13             target_switch_duration_s: 2.0
14             source_port: 9
15             destination_port: -1
16     scheduling:
17         simulation_duration_s: 300.0
18     autonomous_systems_connections:
19         network_address: 10.2.1.0
20         network_mask: 255.255.255.0
```

Listing 4.5 shows the configuration structure for the global settings. The `capture` setting consists of only one property, namely the prefix that will be used during the naming of the PCAP files. The `capture` setting is optional.

The same goes for the `scheduling` setting which defines the total duration of the simulation and the `autonomous_systems_connections`, which enables the specification of the network addresses used to create the connections from ASs to their respective CN node. The reason these network addresses are separate from those of the CN or the AS is that they are entirely 'transparent', *i.e.,* do not show up in any of the PCAP output.

Finally, there are the `attack` settings, which define the characteristics of the pulse-waves. The burst duration and target switch duration can be specified globally (lines 5 and 6) and are optional. The individual attack vectors only require the `type` field, the rest of the properties are optional.

Line 8 shows a minimal vector specification, whilst lines 9 to 15 show a vector specification with all properties defined, with the burst duration and target switch duration on the vector having higher precedence than the global values. The same goes for the source and destination ports where the per-vector values also enjoy greater priority than the per-attacker settings. For both port fields, -1 stands for randomized port, whereas a specific port number fixes the port to that value. Not specifying a port setting means the corresponding value present on the attacker nodes will be used. Naturally, the source and destination port is only used by attack vectors that operate with ports in the headers, so ICMP flooding will ignore these settings.

Listing 4.6: Central Network Configuration

```
1  central_network:
2    network_address: 10.1.1.0
3    network_mask: 255.255.255.0
4    bandwidth: 500Gbps
```

```
5     delay: 1ms
6     degree_of_redundancy: 0.75
7     topology_seed: 35
8     nodes:
9       - id: IXP1
10      - id: IXP2
11      - id: IXP3
```

Listing 4.6 contains an example configuration for a CN. The network mask and network address are optional as they are transparent, only serving for routing purposes but never showing up in any of the PCAP output files. The bandwidth and delay settings are also optional.

The `degree_of_redundancy` controls the amount of superfluous connections within the CN topology. A degree of zero means a minimal topology with `N-1` connections for `N` nodes, whereas a degree of 1 results in a CN topology with `N*(N-1)/2` connections, *i.e.,* the amount of connections that would be found in a full mesh with `N` nodes. The `degree_of_redundancy` setting is also optional.

The `topology_seed` is used to seed the random generator used for building the randomized topology within the CN. A seed is used to ensure that all parallelized instances (when using MPI) end up with the same CN topology. The setting is optional. Furthermore, having the randomization controlled by a configurable seed also ensures the reproducibility of results, contributing to design requirement **R4**.

Finally, the `nodes` list contains the list of CN nodes with only their respective identifier. The identifier is required for each node and there are no other settings available for CN nodes.

Listing 4.7: Autonomous Systems Configuration

```
1  autonomous_systems:
2    - id: AS1
3      network_address: 192.168.1.0
4      network_mask: 255.255.255.0
5      bandwidth: 100Gbps
6      delay: 10ms
7      attachment:
8        central_network_attachment_node: IXP1
9        bandwidth: 100Gbps
10       delay: 3ms
11   - id: AS2
12     network_address: 192.173.1.0
13     attachment:
14       central_network_attachment_node: IXP2
```

Listing 4.7 shows how ASs are configured. Each AS requires an identifier and a network address. The `central_network_attachment_node` is technically optional but would result in a detached AS (*i.e.,* the AS would not be part of the overall topology) and should thus be also considered required. Thus, the configuration of `AS2` starting on line 11 provides an example of a minimal AS specification.

`AS1`'s configuration, starting on line 2 provides an example of an AS configuration that contains all properties. In addition to the network address, the network mask can also be controlled. Further, bandwidth and delay are configurable both for the internal topology of the AS as well as for the connection to the attachment node (lines 9 and 10).

Listing 4.8: Attacker Node Configuration

```
1  attacker_nodes:
2    - id: AN1
3      owner_as: AS1
4    - id: AN2
5      owner_as: AS3
6      data_rate: 1Mbps
7      packet_size: 378
8      source_port: 256
9      destination_port: 80
10     max_data_rate_fluctuation: 0.3
```

Listing 4.8 contains two examples of attacker node configurations. The first (lines 2 and 3) represents the minimal configuration with only the required properties for identifier and owner AS.

The second example, starting on line 4 is an attacker node that has all properties configured. In addition to the minimal configuration, the data rate and packet size are also configurable on a per-attacker-node basis. These data rate and packet size settings have **lower** precedence than the attack-vector specific settings (*cf.* Listing 4.5). The port numbers can also be specified for each attacker node individually as shown in lines 8 and 9. The default values for the ports on the attacker nodes is -1, meaning they are randomized by default. Finally, the maximal data rate fluctuation can be specified. Finally, the fluctuation rate defines the possible range in both directions, *i.e.,* a fluctuation of 0.2 means the actual datarate varies between 0.8 and 1.2 times the originally configured data rate.

It is worth noting, that the `attacker_nodes` key is entirely optional, allowing the simulator to run without any malicious traffic being modeled.

Listing 4.9: Benign Node Configuration

```
1  benign_client_nodes:
2    - id: BN1
3      peer: TN1
4      owner_as: AS1
5      max_reading_time: 100
```

Listing 4.9 shows an example of a benign node configuration. The fields for identifier and owner AS are required and so is the `peer` property, which specifies with which server node (target or non-target) the node establishes communication with. The `max_reading_time` field is optional and is used to set the maximal waiting time before the node fires a new Hypertext Transfer Protocol (HTTP) request towards the peer server. For more information on the maximal reading time refer to Section 4.3.4.1.

The `benign_client_nodes` key, like the one for attacker nodes is optional, enabling the simulator to forego benign traffic and focus solely on malicious traffic.

Listing 4.10: Server Node Configuration

```yaml
target_server_nodes:
  - id: TN1
    owner_as: AS2
    http_server_port: 80

non_target_server_nodes:
  - id: NTN1
    owner_as: AS1
```

Listing 4.10 contains examples for both target and non-target server nodes. Their configuration is structured identically, with the fields for identifier and owner AS being required, whilst the `http_server_port` field is optional.

The target server nodes are attacked by the malicious node in the order in which they are listed in the configuration.

Both the keys for target- and non-target server nodes are entirely optional. It should be noted though, that removing both leads to a situation where no traffic at all is being generated, as malicious nodes rely on target nodes in order to have servers to attack and benign clients require a communication peer from either the target- or non-target server node list.

4.3.2.3 Configuration Validation during Parsing

During configuration parsing, the system performs basic validation of the configured topology, to avoid a situation where topology construction begins with an obviously invalid topology specification. Concretely, the system checks the following settings:

- **Valid attachment node**: Each AS has to specify to which CN node its gateway is supposed to be connected during topology construction. The system, therefore, checks if the configured attachment node is an actual CN node (*i.e.,* if it exists).
- **Valid AS ownership**: Each node type (benign, attacker, target server, non-target server) is required to specify to which AS it belongs. Thus, the system verifies if the specified owner AS exists.
- **Valid peer**: Benign nodes have to specify with which server node (target or non-target, both are valid node pools to draw from) it communicates during the simulation. Hence, the system performs checks to ensure that the specified peer node actually exists.

These validation processes are implemented within the `ConfigurationFileReader`, utilizing a number of different sets to establish pools of valid options for each of the three aforementioned settings, then simply checking if a given specified peer, owner AS or attachment node is present in the respective set. The set container was chosen, as this process does not care about ordering or number of occurrences.

On top of validating the specified topology, the system also checks if required configuration settings are present, throwing errors or warnings (whichever is appropriate) if they are missing.

Listing 4.11: Validating Topology Configuration

```
if (!root.has_child("autonomous_systems"))
{
    NS_FATAL_ERROR("Configuration does not contain any 'autonomous_systems' key.");
}
root = root["autonomous_systems"];
std::vector<AutonomousSystemConfiguration> autonomousSystems;
for (ryml::ConstNodeRef AS : root.children())
{
    if (AS.has_child("id") && AS.has_child("network_address"))
    {
        AutonomousSystemConfiguration ASC(ToString(AS["id"].val()),
                                          ToString(AS["network_address"].val()));
        // parse optionals
        ...
        // parse connection of AS gateway to some central network node
        bool hasAttachmentNodeIdSpecified = false;
        if (AS.has_child("attachment"))
        {
            ryml::ConstNodeRef ASAttachment = AS["attachment"];
            if (ASAttachment.has_child("central_network_attachment_node"))
            {
                hasAttachmentNodeIdSpecified = true;
                std::string nodeId =
                    ToString(ASAttachment["central_network_attachment_node"].val());
                if (m_validCentralNetworkNodeIds.count(nodeId) == 0)
                {
                    NS_FATAL_ERROR("Found AS configuration that references "
                                   "'central_network_attachment_node': "
                                   << nodeId
                                   << ". No such central network node found in the "
                                      "configuration file. The faulty AS
                                      in question: "
                                   << ASC.GetId());
                }
                ASC.SetAttachmentNodeId(nodeId);
            }
            // parse other settings regarding attachment connection
        }
        else
        {
            NS_LOG_WARN("Found AutonomousSystemConfiguration without "
                        "'attachment' key.");
        }
        if (!hasAttachmentNodeIdSpecified)
        {
            NS_LOG_WARN("Found AutonomousSystemConfiguration that does not"
                        " specify a 'central_network_attachment_node' key."
                        " Unless you plan to manually connect the AS in the"
                        " main script, this will lead to an isolated AS that"
                        " is not connected to the main topology.");
        }
```

```
52          autonomousSystems.push_back(ASC);
53          m_validAutonomousSystemIds.emplace(ASC.GetId());
54      }
55  }
```

Listing 4.11 contains excerpts of the `ParseAutonomousSystemsConfiguration` method, which showcases both how valid options are added to a set and how set information is used to determine if a specified option is valid. Additionally, it contains examples of both warnings and errors being thrown if certain settings are missing from the configuration.

The code shown first checks if an `autonomous_systems` property is defined at all. Its absence would mean that no AS has been configured, resulting in a non-functional configuration. Line 3 shows how with NS-3 errors can be thrown that log the error to the console and terminate the program execution.

Lines 6 to 13 show how the list of ASs is iterated upon and for each AS, an AS configuration class is instantiated. Lines 17 to 37 contain code that checks first if any attachment settings are present at all, and then specifically looks for the attachment node property (line 21) and validates it using the corresponding set (line 26), leading to a fatal error if an invalid CN node is specified.

Lines 40 to 51 show examples of warnings being used to communicate the absence of important but not required settings. Finally, on line 53, the AS configuration is considered fully parsed and its own id is added to the set of valid AS ids that can be used in the different node types to specify to which AS a given node belongs.

4.3.3 Topology Construction

Topology construction is composed of three major tasks that serve the goal of ending up with a topology that includes all the configured AS, the CN and the individual nodes within the different ASs. The three tasks are:

1. **Constructing the CN**, resulting in the topology of the network of IXP nodes

2. **Constructing the ASs**, resulting in the topologies of the ASs being connected to the CN and nodes being allocated within the AS to be assigned roles.

3. **Assigning roles to the individual nodes within the AS**, thus having the correct amount of each node type (benign, malicious, target server and non-target server) in each AS.

The relevant details for each of these tasks are discussed in the following sections.

4.3.3.1 Constructing the Central Network

The CN is implemented by a class hierarchy centered around the abstract base class `CentralNetwork`. This base class enforces a minimal set of functions that each descendant needs to implement. Notably, this includes the two methods `EnablePcap` which is used to enable packet capturing (more on that in Section 4.3.6) and `GetNodeById`, which serves the purpose of exposing specific nodes for the purpose of establishing a connection, *e.g.*, when connecting an AS to the CN.

From this base class, the system at current provides two different implementations of a CN, the `FullMeshCentralNetwork` and the `RandomizedPartialMeshCentralNetwork`. The full mesh variant was primarily used during development and is not in use within the actually finished system, as a full mesh topology was seen as too optimistic in terms of what might be encountered in a real world situation, thus is not discussed in this report. Nonetheless, it is present in the repository, should any user wish to make use of it, though that does require making changes to the code.

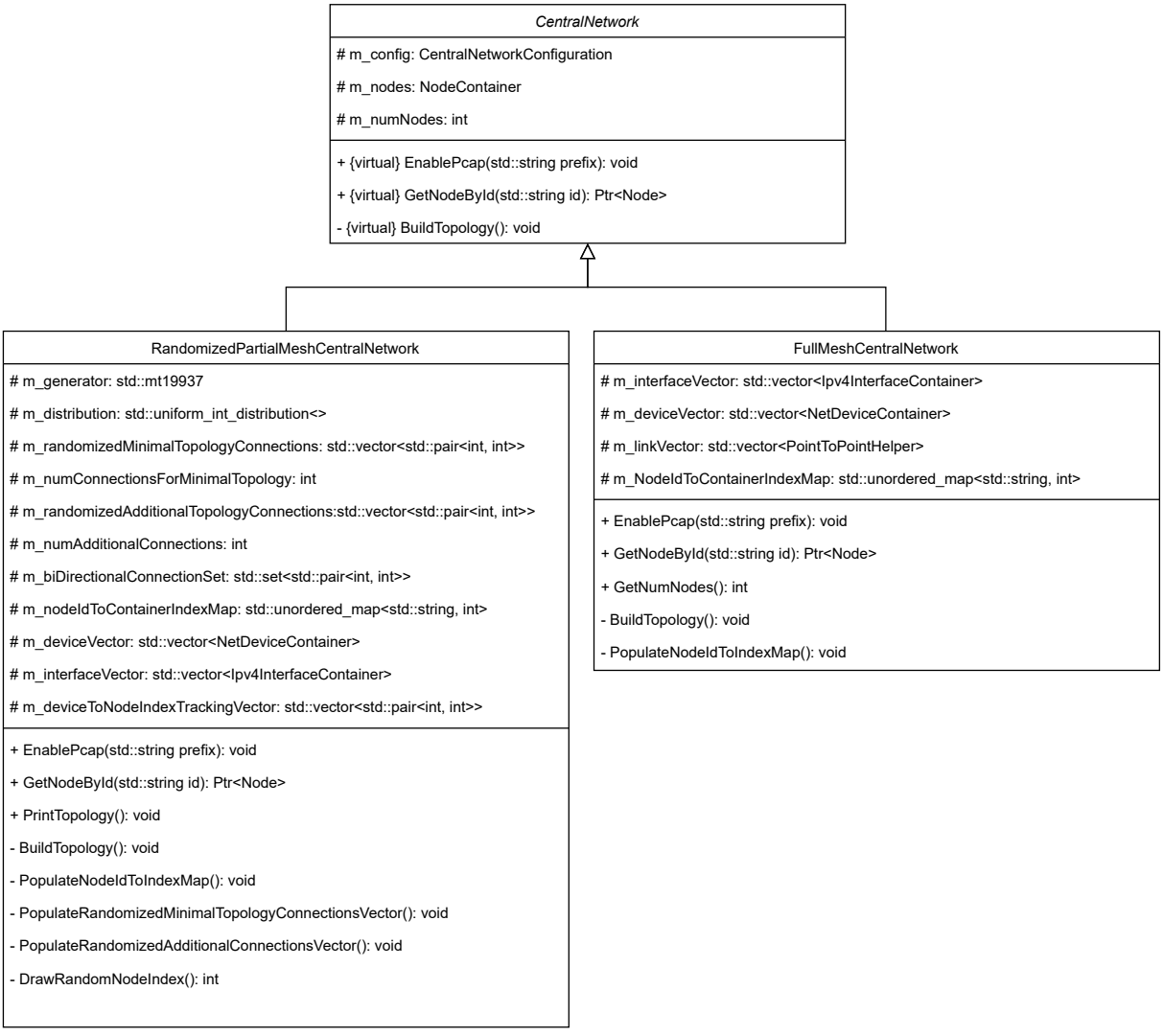


Figure 4.2: Central Network Class Diagram

Figure 4.2 depicts the class diagram for the CN class hierarchy. The full mesh variant was included to highlight how much variety there can be within a CN implementation, ranging from the somewhat simplistic full mesh implementation to the more complex randomized partial mesh variant. The scope of the base class was conciously kept small to allow for a potentially large variety of different CN implementations, thus improving extendability (design requirement **R5**).

Regarding the randomized partial mesh variant, the one actually in use in the current version of the system, there are a number of key aspects that bear further explanation. First, the basics of how a topology is created are explained. Then, the approach to randomization is outlined.

Listing 4.12: Basic Central Network Topology Construction

```
1  NodeContainer nodes;
2  ...
3  // check that number of nodes at least 2 to satisfy point-to-point arithmetics
4  ...
5  nodes.Create(m_numNodes, 0);
6  InternetStackHelper stack;
7  stack.Install(nodes);
8
9  std::vector<NetDeviceContainer> deviceVector;
10 std::vector<Ipv4InterfaceContainer> interfaceVector;
11
12 StringValue dataRate(m_config.GetBandwidth());
13 StringValue delay(m_config.GetDelay());
14 PointToPointHelper p2p;
15 p2p.SetDeviceAttribute("DataRate", dataRate);
16 p2p.SetChannelAttribute("Delay", delay);
17
18 Ipv4AddressHelper address;
19 Ipv4Address addressBase = m_config.GetNetworkAddress().c_str();
20 Ipv4Mask addressMask = m_config.GetNetworkMask().c_str();
21 address.SetBase(addressBase, addressMask);
22
23 for (int i = 0; i < m_numConnectionsForMinimalTopology; i++)
24 {
25     std::pair<int, int> currentPair = m_randomizedMinimalTopologyConnections[i];
26     int firstIndex = currentPair.first;
27     int secondIndex = currentPair.second;
28
29     NetDeviceContainer devices = p2p.Install(nodes.Get(firstIndex), nodes.Get(
       ↪ secondIndex));
30     Ipv4InterfaceContainer interfaces = address.Assign(devices);
31     address.NewNetwork();
32     deviceVector.push_back(devices);
33     interfaceVector.push_back(interfaces);
34     m_deviceToNodeIndexTrackingVector.push_back(std::pair<int, int>(firstIndex,
       ↪ secondIndex));
35 }
36
37 // identical procedure for additional connections
38
39 m_nodes = nodes;
40 m_deviceVector = deviceVector;
```

```
41 m_interfaceVector = interfaceVector;
```

Listing 4.12 shows extracts from the `BuildTopology` method. The first step when building the topology is to create the nodes. For this, the corresponding container is used and the nodes are created in the correct amount and also in the correct system rank (*cf.* Section 4.3.1.1), as seen on lines 1 to 5.

The next few lines (6-21) consist of setting up containers that will hold the relevant parts of the individual point-to-point channels as well as configuring the `PointToPointHelper` used to create the channels and the `Ipv4AddressHelper` used to perform the address assignment.

So far, none of the nodes are connected yet. This is now remedied by iterating through the vector `m_randomizedMinimalTopologyConnections` which specifies which connections need to be established (line 25), such that each node is reachable from each other node, whilst only using the minimal number of connections. For each pair of node indices (referring to the indices within the `NodeContainer` on line 1), a point-to-point channel is created (line 29) and the resulting `NetDevices` are stored (line 32). Similarly, the interface resulting from performing address assignment (line 30) is also stored (line 33).

Then, to ensure that a given pair of NetDevices can be traced back to a specific pair of node indices, a tracking vector is updated (line 34). The same procedure is repeated for the additional connections that go beyond the minimal topology, by iterating through the vector `m_randomizedAdditionalTopologyConnections`. Finally, the containers are stored in the matching class members for access later on (lines 39-40).

Take special note of the `NewNetwork()` call on line 31. This results in each connection residing in its own subnet. Whilst this is an inefficient use of network address space, it is informed by insights gained from preliminary evaluations regarding the behaviour of point-to-point and csma channels. Refer to Section 5.4.0.2 for more information on this matter.

The code shown in Listing 4.12 requires the two vectors `m_randomizedAdditionalTopologyConnections` and `m_randomizedMinimalTopologyConnections` to already be present. The way these two vectors are populated is related although slightly different.

Listing 4.13: Randomization of Minimal Topology Connections

```
1  std::set<int> alreadyDrawn;
2  std::set<int> notYetDrawn;
3  for (int i = 0; i < m_numNodes; i++)
4  {
5      notYetDrawn.emplace(i);
6  }
7
8  for (int i = 0; i < m_numConnectionsForMinimalTopology; i++)
9  {
10     int firstIndex;
11     int secondIndex;
12     bool isInTargetSet = false;
13     while (!isInTargetSet)
14     {
```

```
15          firstIndex = DrawRandomNodeIndex();
16          if (alreadyDrawn.empty())
17          {
18              // set is initially empty, so just accept first draw as is.
19              isInTargetSet = true;
20              notYetDrawn.erase(firstIndex);
21              alreadyDrawn.emplace(firstIndex);
22          }
23          else
24          {
25              if (alreadyDrawn.count(firstIndex) != 0)
26              {
27                  isInTargetSet = true;
28              }
29          }
30      }
31      isInTargetSet = false;
32      while (!isInTargetSet)
33      {
34          secondIndex = DrawRandomNodeIndex();
35          if (notYetDrawn.count(secondIndex) != 0)
36          {
37              isInTargetSet = true;
38          }
39      }
40      notYetDrawn.erase(secondIndex);
41      alreadyDrawn.emplace(secondIndex);
42      m_randomizedMinimalTopologyConnections.push_back(
43          std::pair<int, int>(firstIndex, secondIndex));
44  }
```

Listing 4.13 shows how the minimal topology connections are established. To ensure that all nodes are reachable, a combination of two sets is used. Set `alreadyDrawn` holds all nodes which have already been drawn (*i.e.,* connected to the topology), whilst `notYetDrawn` contains those nodes yet to be connected. The nodes are represented by the index they will occupy within the NodeContainer (*cf.* Listing 4.12, line 1) during topology building.

The set of already drawn nodes is initially empty, whilst the one with disconnected nodes is initially full (lines 1 to 6). Then for the required number of connections (`N-1` connections for `N` nodes), a loop is ran that performs two random draws, one from each set, resulting in a connection that is guaranteed to (1) involve two different nodes and (2) add a new node to the already connected topology. After each iteration of the loop, the node that was newly added to the topology is then removed from the `notYetDrawn` set and moved to the `alreadyDrawn` set (see lines 40 and 41).

Note that the draws on line 15 and 34 actually happen over the entire range of possible node indices (the number of nodes and thus the possible indices are known from the CN configuration), which then has to be checked against the values within the target set (lines 25 and 35 respectively).

Note further, that for the first draw from the `alreadyDrawn` set, special rules apply. As the set is initially empty, the check on line 25 will always fail, therefore when drawing nodes

for the first connection, the draw is simply accepted and the node is moved immediately to the `alreadyDrawn` set to prevent it from being drawn again for the second index on line 34, which would lead to a connection that connects a node to itself, thus not adhering to the minimal topology.

Listing 4.14: Randomization of Additional Connections

```
1  for (int i = 0; i < m_numAdditionalConnections; i++)
2  {
3      int firstIndex = DrawRandomNodeIndex();
4      int secondIndex = DrawRandomNodeIndex();
5      while (firstIndex == secondIndex)
6      {
7          // prevent self-to-self connection
8          secondIndex = DrawRandomNodeIndex();
9      }
10     m_randomizedAdditionalTopologyConnections.push_back(
11         std::pair<int, int>(firstIndex, secondIndex));
12 }
```

When performing draws for the additional, superfluous connections, there is no need to account for nodes not already being connected to the topology. Further, the decision was made to allow for multiple connections between a pair of nodes. Consequently, the logic for drawing these random connections, as shown in Listing 4.14 is significantly simpler, just performing two random draws and ensuring that both drawn node indices are not identical, as that would lead to a self-to-self connection on the given node.

The number of additional connections meanwhile is given by the configured degree of redundancy, as outlined in Section 4.3.2.2. To briefly reiterate, a degree of zero signifies no additional connections, whilst a degree of 1 results in the number of connections as seen in a full mesh topology involving the same number of nodes.

Given that duplicate connections between two nodes are allowed and nodes are drawn at random, a degree of 1 is not a guarantee that the resulting topology is actually a full mesh. Furthermore, one can configure degrees of redundancy higher than 1.

This might make the choice of basing the number of additional connections on that degree seem odd, given that it does not provide any guarantees. However, relying on a degree that depends on the number of nodes seemed preferable compared to having to specify the additional number of nodes in absolute terms. The reason for this is that a degree of redundancy of *e.g.,* 0.5 scales up with the number of nodes, whilst an absolute amount of *e.g.,* 10 additional connections is agnostic to the number of nodes, thus is a lot when the CN only contains 4 nodes, but is very little when the CN contains 40 nodes.

One last fact that needs to be mentioned regarding this CN implementation, is that it requires a modification of the `GlobalRouteManagerImpl`, as you otherwise **may**, based on the randomly generated topology encounter the following error: *assert failed. cond = "m_ecmpRootExits.size () <= 1", msg="Assumed there is at most one exit from the root to this vertex"*. What this error basically states is that it found two equal cost paths from one node in the topology to another and considers this an error.

An investigation of the topic shows multiple threads on the NS-3 community forum regarding this particular error, with [53] in particular being informative and including a solution, which boils down to removing that assertion from the `GlobalRouteManagerImpl`, as well as enabling per-packet ECMP (Equal Cost Multi-Path) routing. Suggestions to remove assertions from working could should in the author's opinion always be regarded with some skepticism, however, this suggestion seems to be at least partially backed up by findings in [32].

Listing 4.15: Modified Section in the Routing Manager

```
1 SPFVertex::NodeExit_t
2 SPFVertex::GetRootExitDirection() const
3 {
4     NS_LOG_FUNCTION(this);
5     // in this project, ASSERT below is removed
6     NS_ASSERT_MSG(m_ecmpRootExits.size() <= 1, "Assumed there is at most one exit from
      ↪ the root to this vertex");
7     return GetRootExitDirection(0);
8 }
```

Furthermore, investigation of the relevant code (Listing 4.15) shows that the method always takes the first vertex (*i.e.,* the one at index 0) anyway as shown in line 7, so there is no harm in having more than one viable vertex available. Therefore, this project includes a modified version of the `GlobalRouteManagerImpl`, with the corresponding installation instructions on the projects repository [61].

Interestingly, the marked comment (comment 16) in [33] suggests that this final version of the system may not actually need to do so, as that error apparently not relevant for point-to-point topologies and, as explained in Section 5.4.0.1, the project switched over from its previously CSMA-based AS implementation to one based on point-to-point channels.

However, the author feels that the modification to the `GlobalRouteManagerImpl` should still be kept, primarily because not doing so would lead to the problem re-emerging once other non-point-to-point implementations of the CN or the AS are added in future work, thus negatively impacting design requirement **R5**. Additionally, it would at least have to be tested if the assert does indeed not fire when using point-to-point topologies, but testing for the absence of something that only occurs randomly is difficult to say the least.

4.3.3.2 Constructing the Autonomous Systems

The construction of the ASs results in having the individual ASs instantiated, connected to their CN node of choice. Further, it results in the ASs having their internal nodes created and connected to their AS gateweay node.

As with the CN construction, a class hierarchy has been created for ASs. Figure 4.3 depicts the AS class hierarchy, with a base class `AutonomousSystem` and two different implementations, one based on point-to-point connections, the other based on a Carrier Sense Multiple Access (CSMA) channel. The figure is kept small to fit the page layout, a larger scale version can be found in the appendix (Figure C.1 in Appendix C).

The CSMA channel was used up until preliminary evaluations showed that there are issues with incomplete traffic (*cf.* Section 5.4.0.1). Consequently, the CSMA variant has been abandoned, but is still available within the repository, should any user wish to experiment with.

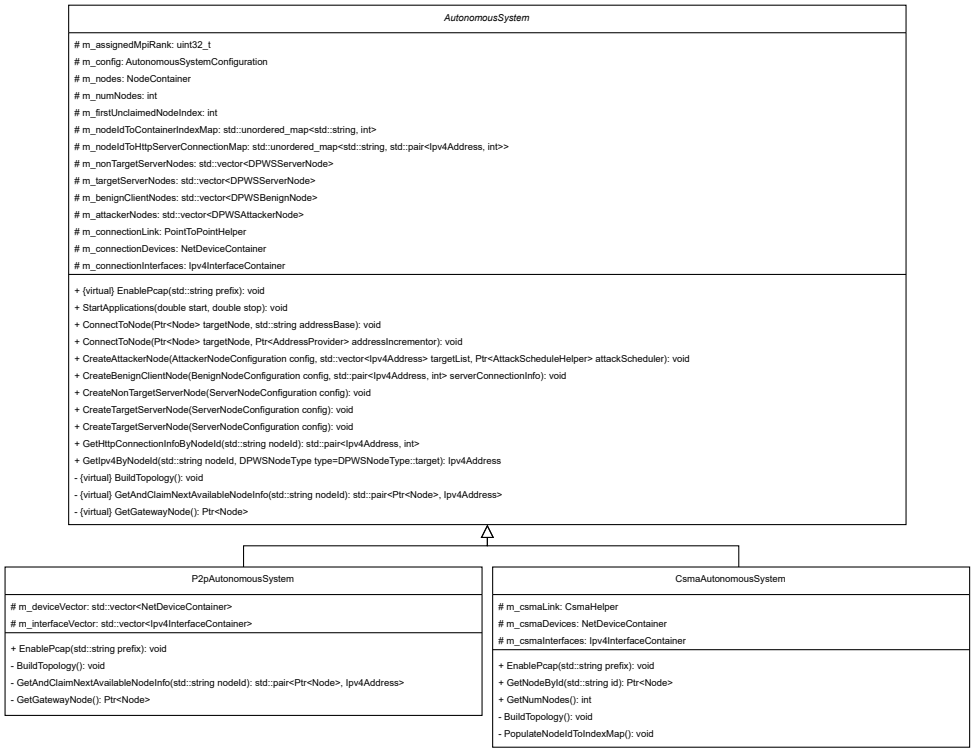


Figure 4.3: Autonomous System Class Diagram

Looking at the class diagram, it is tempting to conclude that a completely different approach was taken than with the CN class hierarchy, where the base class was minimal and much of the logic left to the child classes to implement, whereas the AS base class already contains a lot of logic and implements a large part of the functions directly, instead of having them marked as virtual.

This conclusion would be wrong. The responsibility of a CN is rather different than that of an AS. The CN's main task is essentially just providing a topology through which traffic is being routed. None of the CN nodes install any applications, generate any traffic or are directly addressed in any way by any of the generated traffic.

The AS however has to be able to create different node types, all with their own traffic models. It has to be able to provide information, such as *e.g.,* HTTP server ports or IP addresses about each of its nodes and needs to be capable of issuing start and stop commands to the applications on said nodes. This then explains why the AS base class is so much larger than the CN base class. It's not that the goal of extendability (design requirement **R5**) was abandoned, but rather had to be approached somewhat differently.

In the AS class hierarchy, a large part of the functionality is **not** actually tied to the AS's internal topology, but rather focuses directly on the nodes, without having to know if *e.g.,* the internal topology is realised with CSMA or point-to-point channels. In other words,

the nodes will always be instantiated in a `NodeContainer`, no matter how the topology is ultimately constructed within the AS.

This allows the base class to already implement those parts of the logic that simply rely on nodes being present. As a result, it is the child classes that are rather lean, only having to focus on implementing the internal topology (*i.e.,* connecting nodes and performing address assignments). Thus, extendability is still maintained.

The actual topology construction is largely the same as the one discussed in the context of the CN in Section 4.3.3.1, as the AS implementation is also done using point-to-point channels, except instead of using randomized connections, each node within an AS connects to exactly one node, the AS gateway. Therefore the construction of the AS internal topology is not showed in code.

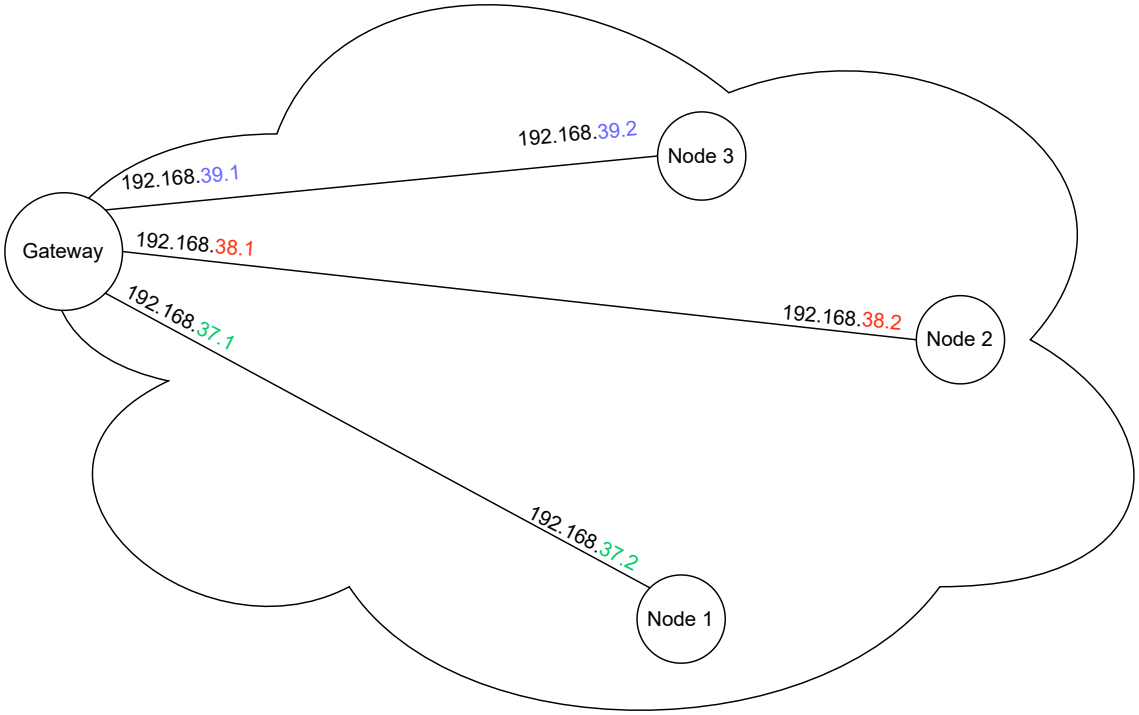


Figure 4.4: Autonomous System Internal Topology

As also already hinted at in Section 4.3.3.1, each point-to-point connection lives in its own subnet, due to routing issues that otherwise occur (*cf.* Section 5.4.0.2). This results in an internal topology within each AS that follows the pattern shown in Figure 4.4, where the AS gateway always is granted the first address and the node the second address for each point-to-point specific subnet.

After the internal topology is constructed, the AS gateway simply needs to be connected to the configured CN attachment node. The procedure is shown in Listing 4.16, where the AS gateway node and the `targetNode`, *i.e.,* the node to which the connection is supposed to be established are connected using a point-to-point channel.

Listing 4.16: Connecting the AutonomousSystem to the Central Network

```
1  PointToPointHelper p;
2  p.SetDeviceAttribute("DataRate", StringValue(m_config.GetAttachmentConnectionBandwidth
       ↪ ()));
3  p.SetChannelAttribute("Delay", StringValue(m_config.GetAttachmentConnectionDelay()));
4
5  NetDeviceContainer devices = p.Install(GetGatewayNode(), targetNode);
6  Ipv4AddressHelper* address = addressProvider->GetAddressHelper();
7  Ipv4InterfaceContainer interfaces = address->Assign(devices);
8  address->NewNetwork();
9  // reuse Ipv4AddressHelper instance provided through argument
10
11 m_connectionLink = p;
12 m_connectionDevices = devices;
13 m_connectionInterfaces = interfaces;
```

The connection is set up with the corresponding settings from the configuration (lines 2 and 3), with an `AddressProvider` instance supplying the `Ipv4AddressHelper` for address assignment purposes. Note that when using MPI, the AS gateway node will potentially be on a different system rank than the node from the CN. This is handled automatically by NS-3 's MPI module, as it will detect that the two nodes are not within the same system rank and then instantiate a so called 'remote' point-to-point channel [55] instead of a regular one.

4.3.3.3 Assigning Roles to Autonomous System Nodes

Once the ASs' internal topologies are constructed and the ASs are connected to the CN, the nodes need to be set up to exhibit the correct behaviours. For this purpose another set of classes was created that rely on being 'assigned' to a given node within an AS and then install the necessary applications that execute the desired behaviours for that node type. Additionally, they provide functions that expose *e.g.,* connection information or provide control over the start and stop timings of the installed applications.

Figure 4.5 shows the class diagram for these 'DPWSNode' classes. The name was specifically chosen to avoid confusion around the 'node' term, as NS-3 has its own node class hierarchy. The prefix 'DPWS', short for Distributed Pulse-Wave Simulator is used to be able to clearly distinguish the two class hierarchies when encountering them within the code.

Each type of DPWSNode 'wraps' around a single NS3Node and represents a specific 'role' that represents the behaviour the node should exhibit. As the name suggests, the `DPWSAttackerNode` is used to implement malicious behaviour, the `DPWSBenignNode` provides benign behaviour and the `DPWSServerNode` is used to implement both target as well as non-target server behaviour.

Each of these classes implements a specific traffic model or at least a part of a traffic model. The discussion of these traffic models is delegated to Section 4.3.4. The malicious node also contributes to the attack scheduling logic, *i.e.,* the logic related to the coordinated

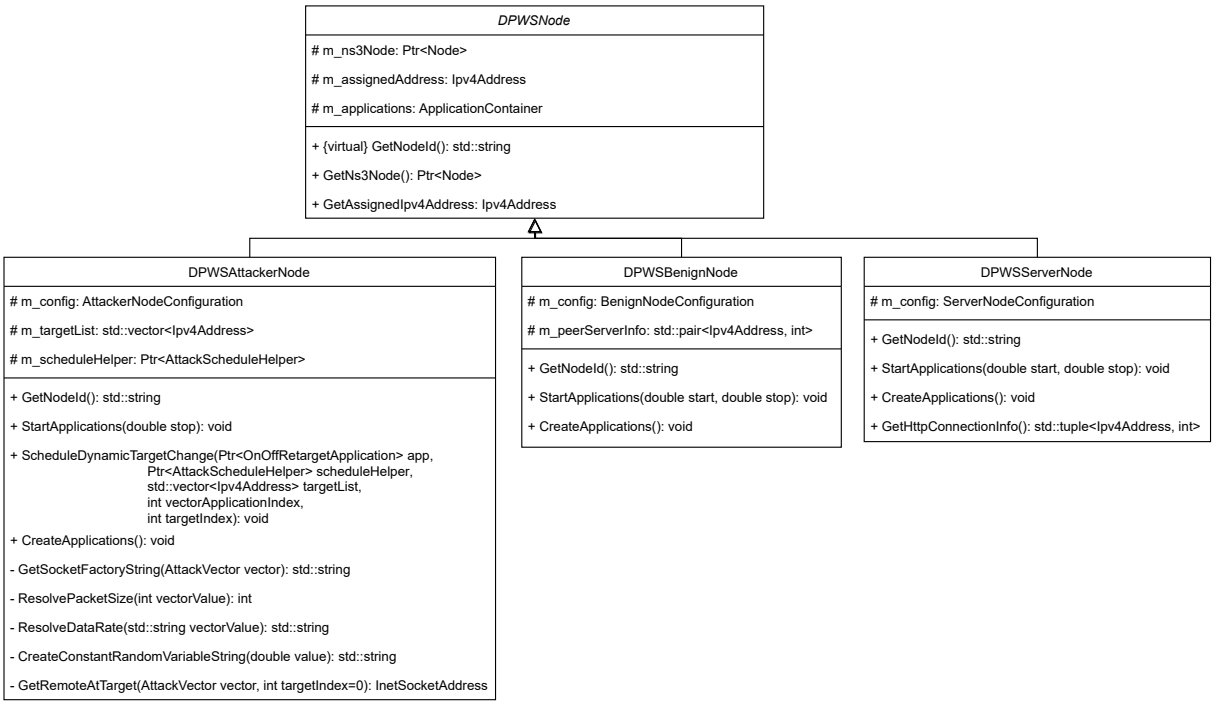


Figure 4.5: DPWSNode Class Diagram

use of specific attack vectors to attack specific targets at specific times. This too is not discussed here and is instead examined in detail in Section 4.3.5.

What is discussed here is how the DPWSNode behaviours are applied to NS3Nodes within the topology. Figure 4.3 shows that each AS provides four methods specifically for that purpose. Despite having different signatures in terms of the arguments they expect, they are structurally identical, with the difference in arguments being due to different DPWSNode types expecting different arguments in their constructor.

Listing 4.17: Creating a DPWS Attacker Node

```
1  // subclass method
2  std::pair<Ptr<Node>, Ipv4Address>
3  P2pAutonomousSystem::GetAndClaimNextAvailableNodeInfo(std::string nodeId)
4  {
5      int index = m_firstUnclaimedNodeIndex;
6      m_firstUnclaimedNodeIndex += 1;
7
8      return std::pair<Ptr<Node>, Ipv4Address>(m_nodes.Get(index),
9                               m_interfaceVector[index - 1].GetAddress(1));
10 }
11
12 // base class method
13 void
14 AutonomousSystem::CreateAttackerNode(AttackerNodeConfiguration config,
15                                      std::vector<Ipv4Address> targetList,
16                                      Ptr<AttackScheduleHelper> attackScheduler)
17 {
18     auto [nextAvailableNode, address] = GetAndClaimNextAvailableNodeInfo(config.
       ↪ GetNodeId());
```

```
19      DPWSAttackerNode aN(config, nextAvailableNode, address, targetList,
          ↪ attackScheduler);
20
21      if (MpiInterface::GetSystemId() == m_assignedMpiRank)
22      {
23          aN.CreateApplications();
24      }
25      m_attackerNodes.push_back(aN);
26  }
```

Listing 4.17 shows how DPWSNodes are assigned to NS3Nodes based on the `DPWSAttackerNode` example. The `CreateAttackerNode` method first 'reserves' the next available NS3Node in the AS topology by calling the method `GetAndClaimNextAvailableNodeInfo` on line 18.

This method returns a smart pointer to the next available NS3Node (*i.e.,* the next NS3Node in the `NodeContainer m_nodes` which has not yet been assigned DPWSNode behaviour and which is not the AS gateway) as well as the IPv4 address it received during AS topology construction (lines 8 and 9).

Take note of the fact that `GetAndClaimNextAvailableNodeInfo` is not part of the AS base class, but rather is specific to a concrete AS implementation, because address assignment is something that varies with the internal AS topology, thus may look quite different for AS implementations that do not rely on point-to-point channels.

Once the NS3Node and address are received in the `CreateAttackerNode`, they are used as arguments during construction of the `DPWSAttackerNode` (line 19). Finally, a check against the AS's assigned system rank is ran to ensure that applications are only installed on NS3nodes if the rank of the parallel instance (when using MPI) matches the one assigned to that AS (lines 21 to 24).

Listing 4.18: Orchestration of DPWSNode Creation in 'main' Script

```
1  ...
2  for (auto aConfig : aNConfigVector)
3  {
4      aSVector[nodeLookupMapper.GetAsIndexByAsId(aConfig.GetOwnerAS())].
         ↪ CreateAttackerNode(
5          aConfig,
6          targetList,
7          &scheduleHelper);
8  }
9  ...
```

The orchestration of these DPWSNode behaviour assignments are part of the 'main script' and is shown in Listing 4.18 on line 4, where for each configured attacker node, first the `NodeLookupMapper` (*cf.* Section 4.3.1.2) is used to get access to the correct AS, and then call the previously discussed `CreateAttackerNode` method.

4.3.4 Traffic Models

The system makes use of two distinct traffic models, one model that implements benign user behaviour and one which is responsible for modeling the behaviour of attackers. As discussed in Section 4.3.3.3, at the end of the topology construction phase, the different DPWSNode types are used to assign behaviours to the individual NS3Nodes within the ASs.

This results in, as Listing 4.17 shows on line 23, the `CreateApplications` method being called for each individual DPWSNode, with each DPWSNode type then installing the appropriate traffic model applications on the corresponding NS3Node. In the remainder of this section, the individual traffic models are explained and the aforementioned installation procedure is discussed.

4.3.4.1 Benign Traffic Model

The purpose of the benign traffic model is to supply a configurable amount of non-malicious background traffic, such that datasets can be generated that contain a mixture attack and benign traffic.

The system relies on an existing implementation of an HTTP traffic generator called 'ns-3-http-traffic-generator' by GitHub user Saulo Da Mata [28]. As stated on the repository page, the implemented model is based on [62], which analyzed the page structure of the most visited website and parametrized their model in accordance with the findings.

This model was chosen for a number of reasons. First, building a benign traffic model from scratch does not only require implementation effort (*i.e.,* writing the actual code) but also needs a theoretical foundation on which to base the model, thus either relying on existing studies of such traffic or alternatively conducting the traffic analysis as part of this thesis. The latter was seen as unrealistic and out of scope due to the time constraints placed upon this project. In the same spirit, if there already exists a suitable implementation of a benign traffic model, then not having to perform the implementation oneself leaves more time to focus on the main goal of the thesis, namely creating a capable and compelling pulse-wave DDoS traffic model.

The second reason for choosing this model was its easy of use, as integrating the model into the existing system simply boiled down to installing applications on the correct NS3Nodes.

Listing 4.19: Installing the Benign Traffic Model

```
1 void
2 DPWSBenignNode::CreateApplications()
3 {
4     auto [peerAddress, peerPort] = m_peerServerInfo;
5     HttpClientHelper httpClient(peerAddress, peerPort);
6     httpClient.SetAttribute("MaxReadingTime", UintegerValue(m_config.GetMaxReadingTime
        ↪ ()));
7     m_applications.Add(httpClient.Install(m_ns3Node));
8 }
9
```

```
10 void
11 DPWSServerNode::CreateApplications()
12 {
13     // DPWSServerNode used for both targetServers and nonTargetServers
14     HttpServerHelper httpServer(m_config.GetHttpServerPort());
15     m_applications.Add(httpServer.Install(m_ns3Node));
16 }
```

The benign traffic model requires a pair of applications to be installed in order to function: An HTTP server application that responds to requests from clients and an HTTP client application that sends HTTP requests to said server and initiate the transfer of data.

Listing 4.19 shows how the installation procedure for both applications looks like. The HTTP Client is installed on `DPWSBenignNode` instances, such that they can initiate a connection with any HTTP server, thus resulting in background traffic flowing through that connection.

Lines 1 to 8 show how this is implemented in code, with the installation being as simple as accessing the IPv4 address and the server port of the HTTP server it wishes to connect to (line 4), then setting up the corresponding helper class (line 5) and configuring the maximum reading time (line 6) and then finally, on line 7, installing the client on the NS3Node and adding the installed application to `m_applications`, which is a `ApplicationContainer`, as can be gleaned from Figure 4.5.

This last step bears further explanation. In principle, one could simply store the HTTP client application directly on some class member to later start it through a call to `DPWSBenignNode::StartApplications`. However, the advantage of instead using a `ApplicationContainer` is that this leaves room for future extension, to *e.g.,* use more than one benign traffic model, which might mean having to potentially install multiple applications on the same NS3Node. Any application start commands are then issued to the NodeContainer, which propagates it to each application stored within, meaning that any additionally added applications receive them automatically as well.

This choice was specifically made with design requirement **R5** in mind and has been applied in the same fashion to all DPWSNode types by already including the `ApplicationContainer` in the `DPWSNode` base class.

Another important fact that needs to be explained is the setting of the maximum reading time on line 7. The maximum reading time refers to the time between a client having fully received a page and requesting the next, basically the time it would take an actual user to read the page [62]. The implementation of the traffic model as found on GitHub has a maximum reading time of 10'000 seconds, as the model formulated by [62] defines it as such, with the number originating in the findings of [44].

The issue with having a maximal reading time of approximately 2 hours and 45 minutes, is that it renders the model difficult to use for simulations with shorter duration. To account for that, the implementation of the HTTP client was slightly changed to allow for the maximal reading time to be configurable, to account for shorter simulations.

Listing 4.19 also shows the installation process for the HTTP server application, which is even more straightforward than the installation of the HTTP client. Lines 10 to 16 show

how the server application is installed on `DPWSServerNode` instances, which are used to model both targets as well as non-target servers. Here too a helper class is first created (line 14) which is then used to instantiate the actual server application (line 15) and add it to the `ApplicationContainer`.

As already hinted at, the HTTP server is installed on all server nodes, no matter if they have been designated as targets for attack traffic or not. This decision was made to allow target servers to receive a mixture of benign and attack traffic or to also have the traffic types be completely separate, by only creating benign traffic between benign nodes and non-target servers, whilst the attack traffic flows between attacker nodes and target servers.

4.3.4.2 Attack Traffic Model

The task of the attack traffic model is to provide means to generate different types of DDoS traffic and to do so in a way that matches the characteristic fast-peaking, burst-like nature of pulse-wave attacks. A distinction, therefore, has to be made between modelling the traffic in a pulse-wave pattern and modelling the makeup of the traffic in terms of the actual DDoS attack that is being conducted in a given pulse.

In earlier parts of the report, the term 'attack vector' has been used to refer to type of DDoS attack (*e.g.,* UDP flooding or ICMP flooding). This terminology is maintained here, with the term 'pulse' specifically referring to the overall traffic pattern, irrespective of the attack vector used in that given pulse.

In order to fully understand how the attack vectors are implemented, the implementation of the traffic pattern has to be explained first. The application responsible for the generation of attack traffic is the `OnOffRetargetApplication`, a custom extension of the `OnOffApplication`, a default application contained within NS-3 .

The regular `OnOffApplication` has capabilities for sending packets of configurable size at a configurable data rate to a single predetermined target address and port. Additionally, it runs an on-off schedule in either randomized or constant intervals in which the application will switch from a sending state to an off state and vice versa.

This made it a good starting point for implementation, as the provided functionality already covers a good portion of what is needed to implement the pulse-wave traffic pattern. The data rate and packet sizes are present, allowing for differently constructed traffic in terms of the overall packet volume sent at a given data rate (larger packets at the same rate result in smaller packet volume compared to smaller packets at the same data rate) as well as just the data rate in general. The on-off scheduling cycle can be used to implement a given pulse within the pulse-wave pattern.

The exploration of the application logic in code form is omitted, as there are numerous intermediate and helper methods involved, which make it impractical to illustrate the behaviour of the `OnOffApplication` in its on and off states respectively. Figure 4.6 shows a simplified diagram that illustrates the application's behaviour.

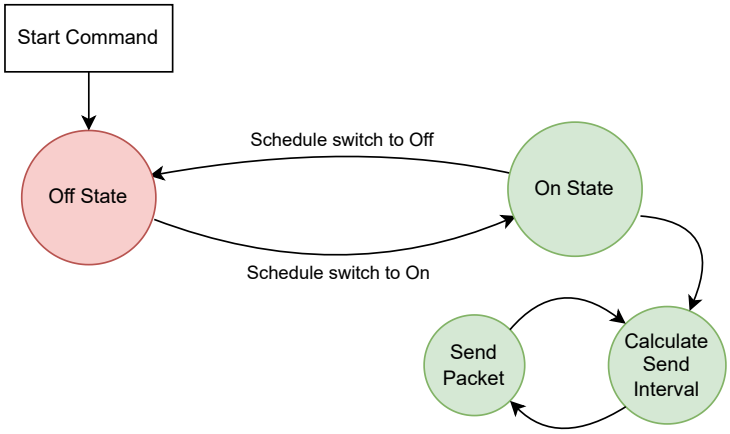


Figure 4.6: OnOffApplication State Cycle

Initially, upon receiving the start command, the application will pass a first off state duration, thus not generating traffic. At the same time, it will schedule a switch over to the on state. Once the simulator schedule has reached that switch over event, the application is set to the on state. It will then immediately schedule the next switch-over to the off state. It also will start sending packets in the following way: the application will calculate the time at which to send the next packet (*i.e.,* calculate the sending interval between packets based on configured data rate and packet size) and schedule a send event. Once that send event is reached, the packet is sent and the next send-even is scheduled by calculating at what time in the simulation it should occur. This cycle is continued until the application switches back to the off state.

What was missing was the ability for the application to switch the target address (and potentially port) as well as the ability to cut out that initial time spent in the off state when the application receives its start command, which is where the `OnOffRetargetApplication` comes into play. The cutting out of the initial off state was easily addressed and is not discussed in this report. The setting of a new target however bears further discussion.

In principle, simply setting a new remote address or port is done easily. When the application is constructed, its socket, used to send packets of the correct type, is instantiated and the connection with the target is established. Setting a new remote then boils down to communicating the new target address and port to the application, which then can disconnect the socket, set the new remote, and then reconnect, with the socket now pointing to the new target.

The topic was broached on the NS-3 community forums by the author, leading to a conversation with one of the NS-3 contributors [52]. The key insight from that conversation was that this will not work for protocols that rely on actually establishing a connection through a handshake procedure, such as TCP [52]. Closing such a connection is not immediate, making it not a viable choice to set a new remote this way [52]. A better approach would thus be to completely close and remove the socket and instantiate a new one with the new target address.

Listing 4.20: Setting a new Remote

```
1  void
2  OnOffRetargetApplication::SetRemote(InetSocketAddress address)
3  {
4      m_socket->Close();
5      m_connected = false;
6      m_peer = address;
7      CancelEvents();
8      ManageSocketCreation();
9  }
10
11 void
12 OnOffRetargetApplication::ManageSocketCreation()
13 {
14     m_socket = Socket::CreateSocket(GetNode(), m_tid);
15     if (m_protocol != -1)
16     {
17         m_socket->SetAttribute("Protocol", UintegerValue(m_protocol));
18     }
19     ...
20     m_socket->SetConnectCallback(MakeCallback(&OnOffRetargetApplication::
       ↪ ConnectionSucceeded, this),
21                                  MakeCallback(&OnOffRetargetApplication::
       ↪ ConnectionFailed, this));
22     m_socket->Connect(m_peer);
23     ...
24 }
```

Listing 4.20 shows the relevant extracts from the code of the `OnOffRetargetApplication`, where the new `SetRemote` method can be used to set a new address and port combination (`InetSocketAddress` contains both the IP as well as the port). Lines 4 to 8 show how this is achieved. First, the old socket is closed and the flag used for internal control flow is set to false (line 5). Then the new peer is written to the class member and all pending events are canceled (send events, as well as scheduled events for switching from on to off or vice versa). In that sense, setting the remote in a way constitutes a complete restart of the application. Finally, the new socket is created (line 8).

Lines 12 to 22 show the relevant parts of socket creation. On line 14, a call to create the new socket with the correct `TypeId` is made (the `TypeId` essentially tells the code which object factory to use, resulting in the correct type of socket). Lines 15 to 18 also show another change made with the `OnOffRetargetApplication`, namely the ability to set the protocol number on the socket. This is relevant if a raw socket type is configured, which is the case for all attack vectors in the prototype.

Once the socket is created, the callbacks (lines 20 and 21) for successful or failed connections are set (which will start the on off cycle, respectively terminate the application), and the socket can be connected to the new remote (line 22). With that, the application now covers all that is needed to implement the desired pulse-wave traffic pattern.

One issue that needs to be addressed at this point is that according to the aforementioned forum conversation, closed sockets are not completely removed from memory, thus essentially introducing a memory leak [52]. Ultimately, this resulted in the NS-3 contributor Tommaso Pecorella creating a fix for that memory issue, which at the time of writing this

part of the report has been merged and released with version 3.39 of NS-3 [60]. This project however was developed, tested and evaluated with version 3.38 of NS-3 , thus this project's repository ([61]) includes the necessary files and instructions on how to apply that memory fix to pre-3.39 versions of NS-3 .

In terms of **attack vectors**, the system implements three distinct vectors, namely 'UDP flooding', 'ICMP flooding' and 'TCP SYN flooding'. These were chosen because [19] specifically mentions them as suitable candidates for pulse-wave attacks. To the best of the author's knowledge, there are no publicly available datasets containing pulse-wave attack data, thus there is no other information to go by regarding what vectors to implement.

Similarly, due to the lack of pulse-wave datasets, the implementation cannot draw from the analysis of real world pulse-wave attacks in terms of how the attack vector implementations should be modelled or behave in terms of the. Thus, in accordance with the thesis goals the approach was chosen to implement them simply based on the description of the attack vector with configurability of parameters such as packet size, data rate and (where appropriate) port numbers allowing for the attack vectors to be tailored towards the signature users wants to utilize in their simulations.

UDP Flooding is the most straightforward to implement. In principle NS-3 already includes a fully functional UDP socket, which when configured as the socket `TypeId` on the `OnOffRetargetApplication` is already capable of producing the desired attack traffic in the form of UDP packets. However, this would make implementing the parametrization of the source and destination ports very complex. A more promising approach, shown to be successful by the EDDD project ([23]) which ran in parallel to this thesis at the CSG at the UZH ([74]), is to forego existing socket implementations and rely on the so-called `Ipv4RawSocketImpl` which gives access to the the protocols without the accompanying logic that is present on protocol-specific socket implementations.

Listing 4.21: UDP Flooding Implementation using Raw Socket

```
1  // part of OnOffRetargetApplication::SendPacket
2  ...
3  else if (m_attackVector == AttackVector::udp_flooding)
4  {
5      packet = Create<Packet>(m_pktSize - m_pktSizeOffsets[AttackVector::udp_flooding]);
6      // create UDP header with correct source and destination port
7      UdpHeader udpHeader;
8      udpHeader.SetSourcePort(GetRandomPort());
9      udpHeader.SetDestinationPort(GetRandomPort(true));
10     packet->AddHeader(udpHeader);
11 }
```

As a consequence of using the raw socket implementation, additional logic, such as fully constructing the packets has to be moved into the application. In some sense this blurs the border of responsibilities between the socket and the application, as the application now has to take care of details such as constructing packet headers. On the other hand, having to allow the application to take attack-vector specific actions in its methods as is the case *e.g.,* during packet construction, opens up a more flexible and less complex way of implementing the sending of specially crafted packets and by that logic also an easier way to implement attack vectors than having to build an entire socket as done

in [11]. Ultimately this cannot be seen as a drawback and also helps simplify future extension (thus contributing to design requirement **R5**), especially as it does not preclude the implementation of attack vectors in the form of a dedicated socket, which remains possible.

In the case of the UDP flooding implementation, using the raw socket implementation means that now the packet header and thereby the source and target ports are defined within the application. Lines 8 and 9 in Listing 4.21 show how the ports are handled, with the `GetRandomPort` method returning a randomized port, unless a specific port number is defined for source or destination port in which case it returns that number.

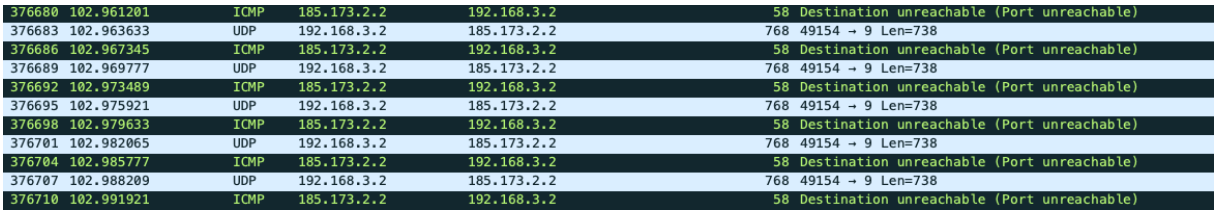


Figure 4.7: UDP Flooding Traffic

The resulting traffic is characterized by UDP packets being sent towards the attack target, with the target responding to each UPD packet with an ICMP packet, informing that there is no process listening on incoming UDP traffic on the target port [38]. How this traffic looks like when reduced to a single attacker and target is shown in Figure 4.7.

The second attack vector is **TCP SYN flooding**. Initially, the goal was to integrate an existing NS-3 implementation of a SYN flood socket, which was produced as the result [11]. That implementation would have fit perfectly with the `OnOffRetargetApplication`, as the only thing that needed to be configured would have been to set the correct `TypeId`, identifying the SYN flood socket and then using the application as normal, meaning the application would be agnostic to the fact that it is executing an attack vector and the responsibility of implementing attack traffic woud rely solely with the socket.

Unfortunately, the author was unable to find the implementation and also was unable of reaching the authors of [11]. An attempt was made to re-implement the SYN flooding socket based on the class diagram provided within [11], but ended up with a non-functional socket factory and was thus abandoned. Instead, once more the EDDD approach of using the `Ipv4RawSocketImpl` was used, which was already proven to be capable of implementing both SYN flooding and ICMP flooding.

The code responsible for implementing the TCP SYN flooding attack vector is structurally very similar to the one shown in Listing 4.21 for the UDP flooding. One thing of note is that SYN packets are implemented as empty packets, thus disregarding any potentially configured packet sizes. Making the packet an SYN packet is not strictly required, as stated in [37], but it still represents the commonly used approach, thus is also done so here.

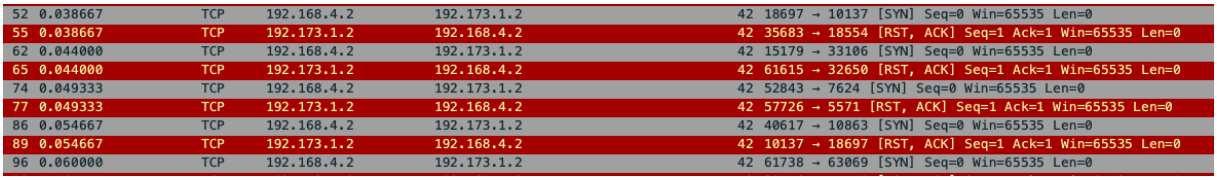


Figure 4.8: TCP SYN Flooding Traffic

Figure 4.8 shows the resulting traffic if reduced to one attacker and one target. Unfortunately, this is not exactly what one would like to see. Instead of returning a 'SYN ACK' flagged packet ([37]), the server simply rejects the connection altogether. What this means is that this attack vector should only be used for simulations where the target's response is not relevant. The project's GitHub page informs potential users of that caveat [61].

Another thing of note about the SYN flood implementation is that it can lead to packets being flagged by Wireshark ([76]) as potential 'TCP Retransmissions' due to port number reuse. This will happen even when randomizing both source and destination port as at some point the combinations are exhausted. When configuring both ports to a specific value then all except the first packet from each attacker IP address will be marked as potential retransmission.

This may be problematic for some analyses, although the SYN flooding dataset by StopDDoS ([29]) also shows packets flagged as retransmissions due to port reuse (*cf.* Figure 4.9).

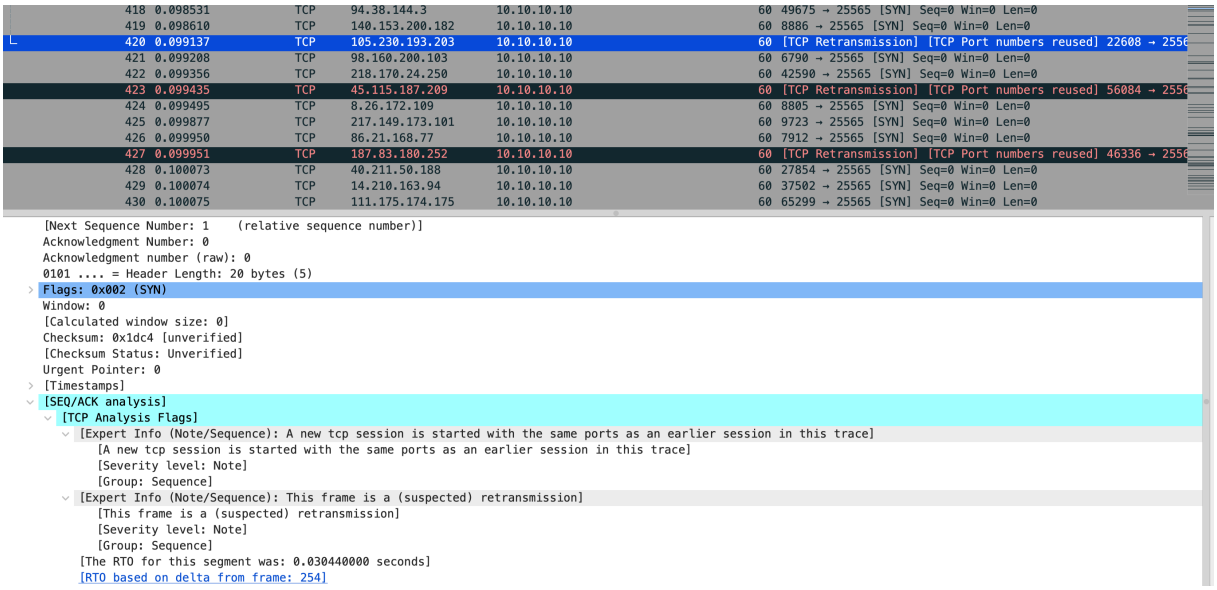


Figure 4.9: Suspected Retransmission in StopDDoS' SYN flood traces [29]

As such the SYN flooding implementation should still be considered viable and is therefore included in the final version of the prototype.

The final attack vector the system implements is **ICMP flooding**. ICMP flooding also makes use of the EDDD project's approach of using the raw socket [23]. In that sense, it too is structurally quite similar to the other two previously discussed attack vectors.

Listing 4.22: ICMP Flooding Implementation using Raw Socket

```
1  // part of OnOffRetargetApplication::SendPacket
2  ...
3  else if (m_attackVector == AttackVector::icmp_flooding)
4  {
5      Ptr<Packet> dataPacket =
6          Create<Packet>(m_pktSize - m_pktSizeOffsets[AttackVector::icmp_flooding]);
7      Icmpv4Echo echo;
8      echo.SetData(dataPacket);
9
10     packet = Create<Packet>();
11     packet->AddHeader(echo);
12
13     Icmpv4Header header;
14     header.SetType(Icmpv4Header::ICMPV4_ECHO);
15     header.SetCode(0);
16     header.EnableChecksum();
17
18     packet->AddHeader(header);
19 }
```

Listing 4.22 shows the part of the `OnOffRetargetApplication` responsible for implementing the ICMP flooding attack. First, on line 5, the data packet is created using the configured packet size, corrected for overhead that stems from packet headers. This data packet is then used as the echo part of the ICMP packet, and added to the 'main' packet (lines 7 to 11).

Once done, the ICMP packet header is constructed with the echo flag, then the 'Code' is set to 0. This relates to ICMP type numbers, as listed in [35], with the echo flag set on line 14 indicating that this is a 'type 8' ICMP packet (*i.e.,* an echo packet) and code 0 meaning that 'no code' has been set, as type 8 ICMP packets do not support (and also do not require) any additional codes.

As a last modification to the header, checksums are enabled on line 16, which in the case of this system means that in the resulting PCAP output the ICMP packets don't carry the '`Checksum: 0x000 incorrect, should be {some-valid-checksum}`' annotation. Then the header is attached to the packet (line 18) and the packet is ready to be sent.

```
102 0.087920   ICMP  192.168.4.2  192.173.1.2  128 Echo (ping) request  id=0x0000, seq=0/0, ttl=61 (reply in 112)
112 0.093680   ICMP  192.173.1.2  192.168.4.2  128 Echo (ping) reply    id=0x0000, seq=0/0, ttl=63 (request in 102)
122 0.098160   ICMP  192.168.4.2  192.173.1.2  128 Echo (ping) request  id=0x0000, seq=0/0, ttl=61 (reply in 127)
127 0.103920   ICMP  192.173.1.2  192.168.4.2  128 Echo (ping) reply    id=0x0000, seq=0/0, ttl=63 (request in 122)
132 0.108400   ICMP  192.168.4.2  192.173.1.2  128 Echo (ping) request  id=0x0000, seq=0/0, ttl=61 (reply in 157)
157 0.114160   ICMP  192.173.1.2  192.168.4.2  128 Echo (ping) reply    id=0x0000, seq=0/0, ttl=63 (request in 132)
182 0.118640   ICMP  192.168.4.2  192.173.1.2  128 Echo (ping) request  id=0x0000, seq=0/0, ttl=61 (reply in 187)
187 0.124400   ICMP  192.173.1.2  192.168.4.2  128 Echo (ping) reply    id=0x0000, seq=0/0, ttl=63 (request in 182)
```

Figure 4.10: ICMP Flooding Traffic

Figure 4.10 shows the resulting traffic, once reduced to a single attacker and a single target. Echo packets are received at the recipient and the response is returned in form of the echo, as is described in [36].

One final aspect of the attack traffic generation must be discussed which is that in an NS-3 simulation, the application will have its data rate locked to the exact value as configured,

leading to inter-packet intervals that are completely constant. Such ideal conditions of operation are likely not reflective of the real world. As such, a per-packet data rate fluctuation is introduced to produce a more dynamic inter-packet interval. This is done in a way that corresponds to the approach taken so far with the prototype and aims to be generic with the ability to tailor the behaviour through configuration, with the option to have specify a separate deviation on each individual attacker node.

Listing 4.23: Packet Interval Randomization with Uniform Distribution

```
1  double
2  OnOffRetargetApplication::GetRandomSendDelayModifier() const
3  {
4      std::random_device dev;
5      std::mt19937 gen(dev());
6      // set range to twice the max, given that it is in either direction
7      std::uniform_real_distribution<> distribution(0, 2 * m_maxDataRateDeviationPercent
         ↪ );
8      // subtract half the range such that both negative and positive values are
         ↪ possible
9      double delayModifierPercent = distribution(gen) - m_maxDataRateDeviationPercent;
10     return delayModifierPercent;
11 }
```

Listing 4.23 shows how this is achieved, by using a fresh random device draw to seed a generator (lines 4 and 5) and then defining the range of possible values the distribution draws from to twice that of the maximally configured deviation. This is done because the data rate can deviate in both directions such that the actually configured data rate value still holds true on average. A value is drawn (line 7) and then adjusted by half the range such that both negative and positive modifiers values are possible (line 9).

For each send event that is scheduled (*i.e.,* for each packet) such a modifier is drawn and applied to the calculated inter-packet interval. It is important to note that the choice of distribution represents a pragmatic choice and is not derived based on analyzing existing datasets. The reasons for that are twofold. The first is that there are not existing pulse-wave datasets so the values would have to be derived from non-pulse-wave DDoS data which suffers from a significant ramp up at the start of the attack, thus the resulting distribution would likely not be suitable for pulse-wave traffic. The second is that a more sophisticated model that *e.g.,* does not operate locally on each individual attacker node and instead attempts to model data rate fluctuations more globally such as for example per network or per connection in the topology is not appropriate to the scope of the thesis and the corresponding time limit.

4.3.5 Attack Scheduling

So far, what was discussed in terms of implementation allows the system to be instantiated and held together by the main script, the configuration file to be parsed and validated, the topology to be constructed based on the configured ASs and the CN, and the desired behaviours and traffic models to be applied to the individual nodes within the topology.

What is still missing to get a functional pulse-wave traffic generator is the coordination of the attacker nodes, ensuring that they execute the correct attack vector in the correct pulse at the correct time and aimed towards the correct target server node. This is where the process of attack scheduling comes into play.

The heavy lifting in this regard is done by the `AttackScheduleHelper` class, with some auxiliary logic also being present in the `DPWSAttackerNode` instances. The task of the `AttackScheduleHelper` is to calculate the schedule at which applications are turned on and off, when they are started and when they need to have their target changed by setting a new remote (*cf.* Section 4.3.4.2).

There is only a single instance of the `AttackScheduleHelper` used within the system, and it is shared by all `DPWSAttackerNode` instances, which ensures that they all run on the same schedule and also avoids having to re-compute the schedule in each individual attacker node. The attacker nodes upon receiving the attack schedule then apply the calculated on and off timings to their `OnOffRetargetApplication` instances and schedule the respective remote changes.

This process is difficult to show in code, thus the exploration of the code is left to those readers curious about the technical details. Instead, in this report the process is demonstrated based on an example, shown in Figure 4.11.

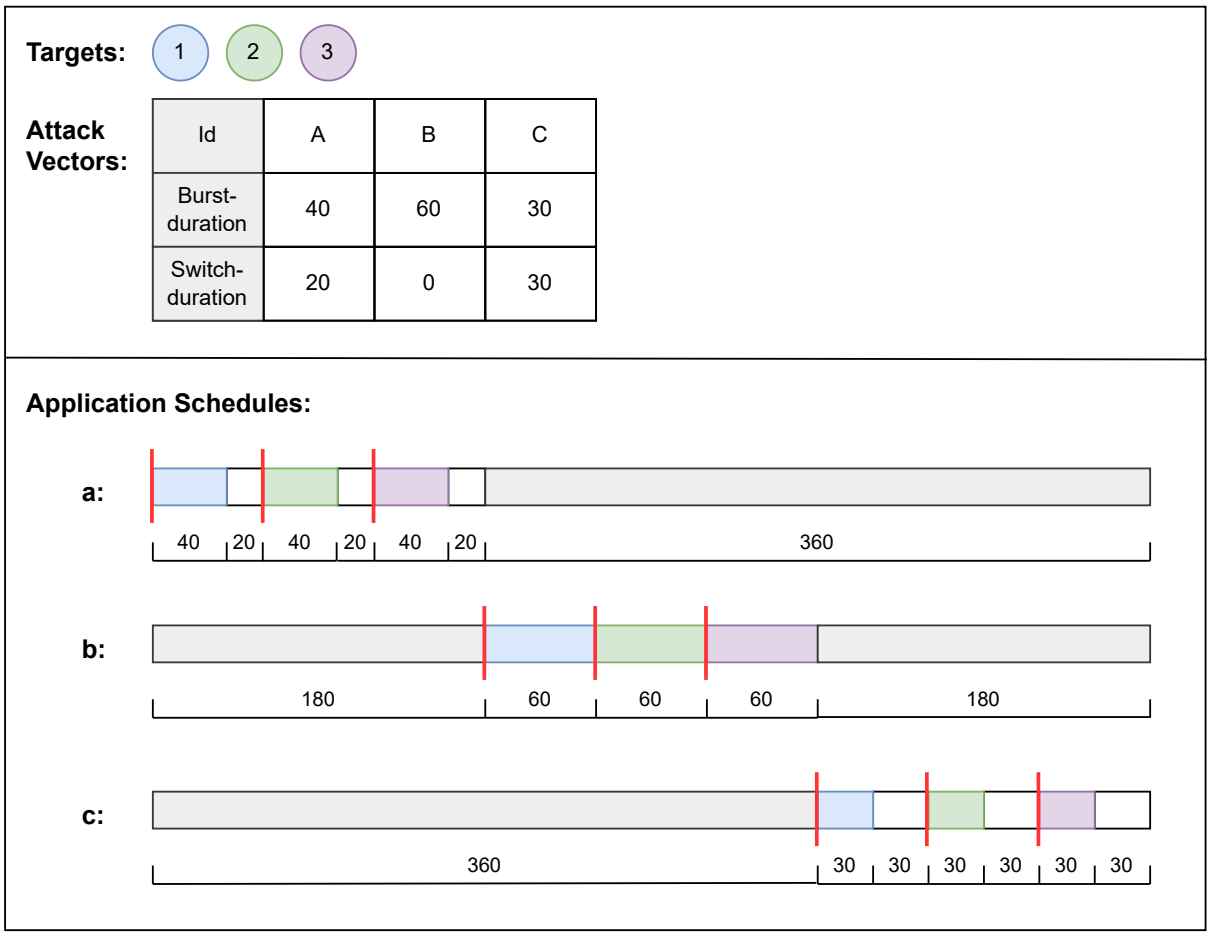


Figure 4.11: Attack Scheduling Example

The example presented in that figure consists of three targets being attacked in sequence, with the attacker nodes using three different attack vectors during the attack. In the table shown within the figure, each vector is listed with its burst duration and its switch duration (*i.e.,* the time it takes for the attack to be redirected to the next target).

Note that switching duration can and typically should be set to 0 (as is the case with attack vector B) as the attacking resources realistically do not cease operation but are just directed towards the new target. The reason the switching duration is configurable is that this way pulse-wave attacks can be simulated against a single target by using the switching duration to mimic the attack resources being directed to somewhere else and then returning to the target.

Each attacker node creates one `OnOffRetargetApplication` instance **per attack vector**, resulting in this example in each attacker node having three separate applications in their application container. In the figure this is represented through the labels, *i.e.,* attack vector 'A' is implemented by application 'a' and so forth. In order to instantiate these applications, the attacker nodes need to know the schedule, which they can access on the `AttackScheduleHelper`.

The `AttackScheduleHelper` is given the number of targets, as well as the configurations of the different attack vectors and based on that creates a schedule for each individual application.

In the figure, the schedule for each application is shown, with the colored blocks representing the application being in their on-state and sending attack traffic toward the target that matches the color of the block. The white blocks in the timeline meanwhile represent the target switching duration, during which the application is in the off-state, thus not producing any traffic.

The grey sections of the schedule also represent the application being in the off-state, but not due to the target switching duration, but rather due to having to allow the other application to perform their round of attacks before it cycles back to the initial application. Lastly, the red bars represent the setting of a new (or initial) target. The schedules shown constitute one 'cycle', *i.e.,* until each application has attacked each target exactly once. These cycles are repeated continuously for each application until the simulation stops.

For application **a** this then means that as soon as the simulation starts, it spends 40 seconds attacking target 1, then spending 20 seconds in off-state before being forced onto target two and later target 3 where for each of them 40 seconds is spent sending traffic and 20 seconds is spent in off-state. Once all targets have been visited, the application spends 360 seconds waiting until the other applications have performed their attacks.

For applications **b** and **c** this is not quite as straight forward, as they cannot directly start sending once the simulation starts, as otherwise all attack vectors are active at once instead of in sequence. To account for that the scheduler calculates an 'application start offset' for each application, which dictates how long the application should wait before it starts up its scheduling cycle once the simulation has began. For application **a**, this offset is 0, for application **b** it is 180 seconds (*i.e.,* the amount of time it takes for application a

to finish), and for application **c** it is 360 seconds (the time it takes for applications a and b to finish).

On a technical level there are a few more details to be handled. The first concerns the setting of the off time duration. Looking at the schedules shown in Figure 4.11 it becomes apparent that there are two different off time intervals. When a given application is attacking any target but the last, the time spent in off-state is shorter than the off-state duration after having attacked the last target.

For example, for application **c**, after attacking targets 1 and 2, it spends 30 seconds in the off-state, whereas after attacking target 3 it spends 390 seconds in off state. The numbers for application **a** are 20 seconds and 380 seconds respectively, with application **b** operating with 0 and 360 second durations.

In order to avoid having to re-configure the applications' off times continuously, the applications are configured with the longer of the two off-duration values, which is calculated such that it also accounts for only one attack vector, hence only one application being present: $off\_time = cycle\_duration - (N * burst\_duration + (N-1) * switch\_duration)$, with N being the number of targets. Applied to the example at hand, this results in 380 seconds for application **a**, 360 seconds for application **b**, and 390 seconds for application **c**, the values already established before.

To still be able to have only the short off-state duration play out after attacking all but the last target, the switching of the target (signified by the red vertical bars) does not only set a new target but also resets the application's on-off cycle, thus effectively forcing it back into the on-state regardless of how much time it still had left to spend in the off-state.

This then leads to the final problem, defining the points in time at which the target switch is to be scheduled. To this end, the `AttackScheduleHelper` uses a pair of indices to track which vector is currently active and which target is currently being attacked, thus essentially having a complete view of where each application currently resides within its schedule, therefore being able to supply each application with the appropriate interval for when the next remote change needs to be scheduled.

4.3.6 Traffic Capture

Regarding the capturing of traffic in PCAP files, NS-3 provides built-in methods to do so, meaning they simply need to be called on the correct locations within the topology, such that the traffic is captured in a way that implements the generation of data from a distributed perspective (design requirement **R1**).

In order to have that distributed perspective implemented, traffic is captured at each interface of each CN node. This provides complete traceability of packets across the entire CN, starting at the AS it originates from all the way to the target AS. To do so, it is sufficient to simply call the `EnablePcap` method on the `NetDevice` instances that belong to said interfaces.

The tricky part is to name the files in such a way that makes it clear where in the topology a given file originates. The file names ultimately take the following form: `{globalPrefix}__{FromID}-to-{ToID}____{junk}.pcap`.

The `globalPrefix` is configurable within the configuration file, such that the user has a way to differentiate files that stem from different simulation runs, by altering said prefix.

The `ToID` is the id configured by the user for the CN node, or, if the file is captured on a CN interface that belongs to a connection stemming from an AS gateway, the id configured by the user for that AS. The `FromID` is always the id of a CN node.

Lastly, the `junk` part represents the part of the file naming done by NS-3 and typically consists of two number, whose meaning the author was not able to fully grasp. Unfortunately, the `EnablePcap` method as present on the `NetDevice` does not seem to provide an option to not have NS-3 add its own part to the file name, hence the insertion of the underscores to clearly separate it from the semantically meaningful part of the file name.

To help illustrate this more clearly, a few examples: If a file is captured on an interface of a connection that connects an AS to a CN node, then the file will be called something along the lines of `testRun__IXP2-to-AS3____{junk}.pcap`. If a file on the other hand is captured on a connection between two CN nodes, then there are two options, either `testRun__IXP2-to-IXP1____{junk}.pcap` or `testRun__IXP1-to-IXP2____{junk}.pcap`.

The reason why there are two options when traffic is captured between two CN nodes, is that the perspective of each CN node must be available in full, such that statements can be made about the traffic as seen on each individual CN node. As a consequence, the traffic is captured on both interfaces of that connection, resulting in two files with mirrored names, with the one where the id of the node whose perspective one wishes to examine comes first belonging to that node.

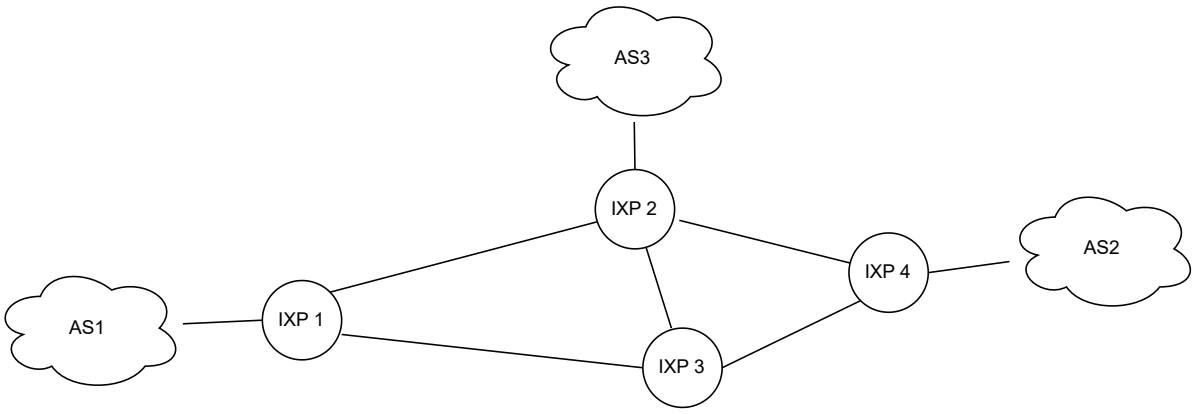


Figure 4.12: Example Topology

To give a more complete example, the assumed topology of some simulation run is shown in Figure 4.12. The resulting PCAP file names, reduced to the `FromId` and `ToId` for each involved CN node are shown in Table 4.1.

As is apparent when looking at the table, in order to get the complete traffic view of a given CN node, it is sufficient to simply look at all the files that have that paritcular CN Node's id as the `FromId`.

Table 4.1: Example Topology File Names

CN NodeId	IXP1	IXP2	IXP3	IXP4
Associated Files	IXP1-to-AS1, IXP1-to-IXP2, IXP1-to-IXP3	IXP2-to-AS3, IXP2-to-IXP1, IXP2-to-IXP3, IXP2-to-IXP4	IXP3-to-IXP1, IXP3-to-IXP2, IXP3-to-IXP4	IXP4-to-AS2, IXP4-to-IXP2, IXP4-to-IXP3

Chapter 5

Evaluation

In this part of the report, the system is put through different evaluation scenarios to test or show different aspects of the prototype. The configuration files used to perform the evaluations are all available on the project's GitHub repository page ([61]).

Regarding the rest of this chapter, in Section 5.1, a set of small scale scenarios is ran to demonstrate the variability in attack vectors that can be achieved through the application of configuration. Then, the distributed nature of the generated dataset is demonstrated in Section 5.2. Afterwards, in Section 5.3 the performance and scalability of the system are evaluated.

This is followed by Section 5.4, where findings that have had an impact on the evaluation and required parts of the system to be changed are presented. These changes have been made before the final evaluation. Nonetheless, they represent valuable takeaways and are therfore included in the report.

Finally, the findings of the evaluation and their implications are discussed in Section 5.5.

5.1 Attack Vector Variability

The purpose of this part of the evaluation is to demonstrate the prototype's ability to generate attack vectors that exhibit different characteristics regarding packet size, packet volume, data rates, protocol, source port and destination port. To that end, in Section 5.1.1, a configuration, **VAR1**, is used that utilizes a combination of per-attacker and per-attack-vector configurations to create those different pulses.

Afterwards, a different configuration, **VAR2** is used to showcase how the duration and overall traffic pattern of the pulses can be controlled. This is done in Section 5.1.2.

5.1.1 Variable Attack Vector Composition

Figure 5.1 shows the result of **VAR1** as a plot of the produced attack traffic expressed in data rate over time. The plot uses colors to indicate the protocol used in the attack vector of a given pulse, with blue representing TCP, green signifying UDP, and brown tones representing ICMP.

The different variations of a given color are used to show which of the 5 attacker nodes contributes which part of the overall traffic, whilst the red line plot indicates the overall packet volume.

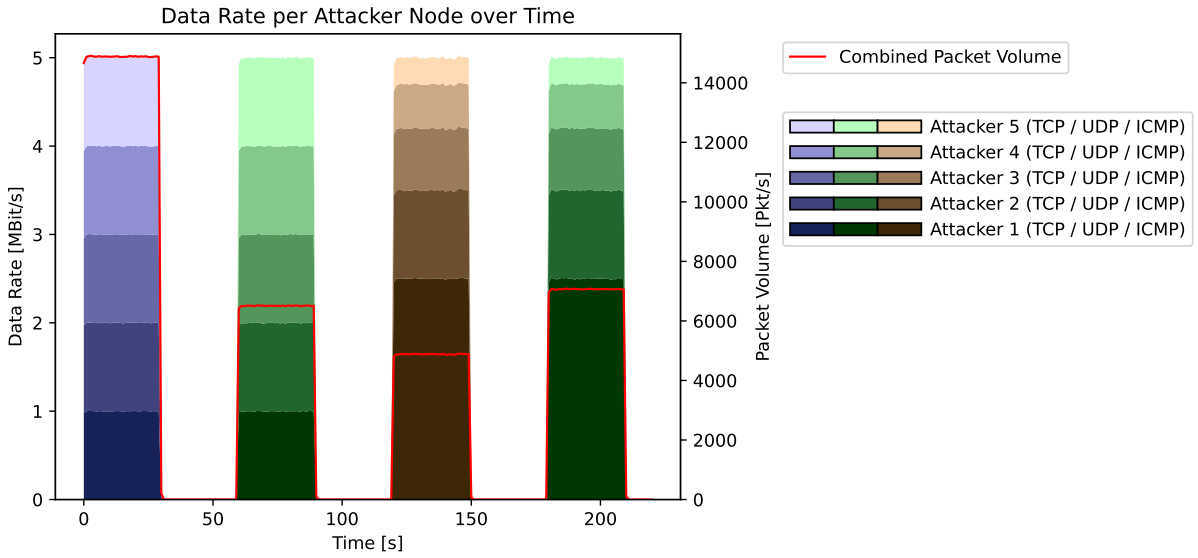


Figure 5.1: VAR1: Data Rate Breakdown per Attacker Across Vectors

Figure 5.2 also shows **VAR1** but with the packet volume being broken down by attacker and protocol rather than the data rate and the red line plot signifying the combined attack traffic data rate. Both figures taken together allow for a more detailed breakdown of the individual pulse-waves.

Both figures have been produced with the `plot_attacker_and_protocol_keyed_traffic_at_ixp_node.py` script, which can be found in the project's repository as well as in the appendix (Appendix D.1).

With the two figures in mind, one can derive facts about the configuration of the individual pulses' attack vectors. All of them have been configured such that the data rates of the individual attacker nodes combined arrive at the same total data rate of 5 Mbit/s, resulting in an average data rate of 5 Mbit/s for each pulse. However, the four different attack vectors go about this in different ways.

Besides the obvious difference in the protocols used, the first and second vectors appear to have the same internal structure, with all attacker nodes contributing equally in terms of data rate. However, looking at the breakdown of the packet volume contributions it

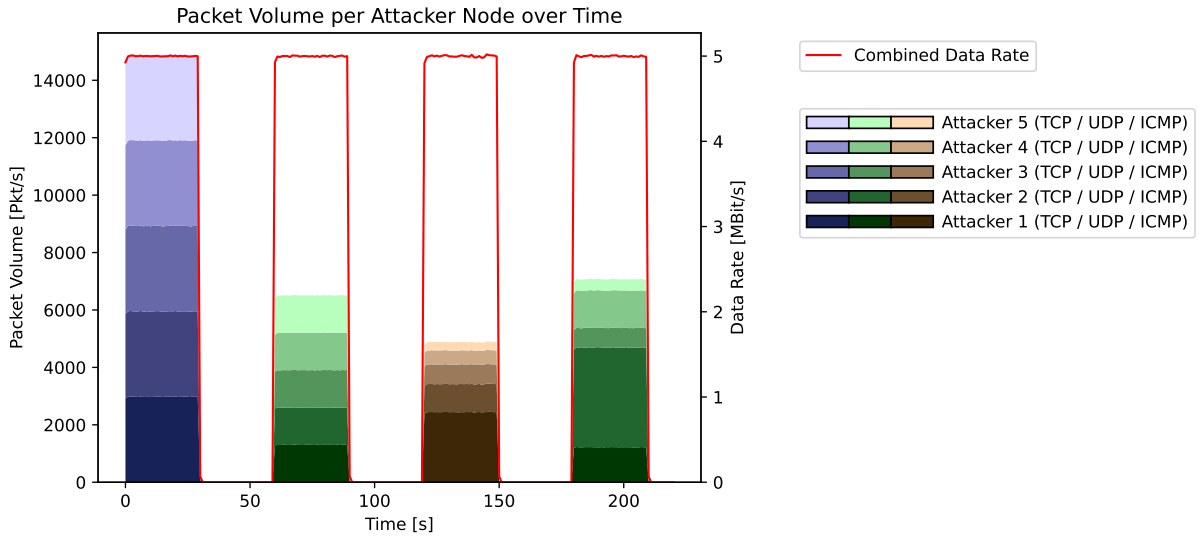


Figure 5.2: VAR1: Packet Volume Breakdown per Attacker Across Vectors

becomes apparent that the first attack vector, using TCP SYN flooding achieves a significantly higher packet volume than the second attack vector (which uses UDP flooding) at the same data rate.

This difference boils down to packet size, with TCP SYN flooding using small, fixed size packets, due to SYN packets being essentially empty packets, whilst UDP flooding packets are freely configurable in their size. In **VAR1**, the second vector uses a packet size that is roughly double that of the fixed size SYN packets, hence the difference in packet volume.

Nonetheless, in the second vector the individual attacker nodes still contribute equally in terms of packet volume as they are all configured to produce packets of the same size. Moving the focus onto the packet volume breakdown of the third vector it may be tempting to state that this has now changed, given that the contributions of the individual attacker nodes to the combined packet volume are no longer equal.

This statement would however be wrong, as the difference in contribution to the overall packet volume is caused not by using different packet sizes, but rather by using different **data rates**. This becomes clear once one considers the data rate breakdown of the third vector as shown in Figure 5.1, where the difference in data rate of the individual attackers is shown. Looking more closely at the packet volume breakdown of the third vector back in Figure 5.2, the ratios between the contributions of the attackers is identical to the ratios seen in the data rate breakdown plot, thus hinting at the fact that the attackers all use the same packet size in vector three.

To elaborate further on this: the relationship between data rate and packet volume, as implemented in the basic `OnOffApplication`, as well as the `OnOffRetargetApplication`, which is what the prototype uses to run the attack vectors (*cf.* Section 4.3.4.2) is in essence given by a three component formula, with packet size being the third part of that calculation.

Consider the way the interval between packets is calculated in those applications:

$$interval = packet\_size/data\_rate$$

Knowing this interval, one can now compute the packet volume that is generated like so:

$$packet\_volume = 1/interval$$

Combining the two equation and rearranging leads to:

$$packet\_volume = data\_rate/packet\_size$$

This is also the formula used by [13]. If the packet size is fixed, then *e.g.,* attacker 1, who contributes approximately 50% of the data rate in the vector must as a consequence also be contributing 50% of the packet volume.

The system, however, also allows for the packet size to be varied across the different attacker nodes. To illustrate this, the focus is now put on the last of the four attack pulses, where *e.g.,* attacker 1 still makes up approximately 50% of the data rate, but now contributes in a reduced manner to the combined packet volume of that vector, whereas attacker 2, who produces fewer data rate than attacker 1 contributes significantly more to the combined packet volume than attacker 1.

This kind of configurable diversity in attack vector composition is seen as **crucial** for this system, as it allows the user to create a wide range of different scenarios by using a mixture of per-attack-vector, per-attacker-node, and global configuration options. For example, pulse-wave attacks are seen as sophisticated DDoS attacks, which require precise orchestration of attack resources and thus are according to [39] likely being executed using small numbers of dedicated, high-capacity devices (*cf.* Section 2.1.1.3).

This then would in the author's view likely result in an attack vector composition where attackers contribute in precisely defined ratios, which is represented by the first two attack vectors in **VAR1** and is achieved by relying on configuration options that apply globally or to specific attack vectors rather than individual attacker nodes.

This, however, does not necessarily exclude the possibility that pulse-wave attacks are performed through the use of an IoT botnet, where the resulting attack vector composition is likely less regular and more varied across the individual attackers, thus the third and fourth vectors in **VAR1** seem to be more appropriate representation of such an IoT based pulse-wave attack, which is achievable by relying more on the system's per-attacker-node configuration options rather than global or attack-vector based ones.

Regardless, the configurability of the parameters data rate, packet size and therefore indirectly packet volume (*i.e.,* packets per second) on both a per-attacker and per-attack-vector basis demonstrates the ability of the system to produce datasets that exhibit highly variable with regard to said metrics. Additionally **VAR1** also demonstrates the ability to configure attack vectors that utilize different protocols. This all contributes towards the main goal of the thesis of being able to configure the system such that a wide range

Table 5.1: VAR1: Vector Characteristic Breakdown

	Attack Vector 1	**Attack Vector 2**	**Attack Vector 3**	**Attack Vector 4**
Packet Sizes (Bytes)	42	96	128	36 (49.1%) 48 (18.4%) 96 (5.5%) 128 (9.7%) 256 (17.3%)
Avg. Data Rate (Mbps)	4.99	5.00	5.00	4.99
Avg. Packet Volume (Pps)	14'874.3	6'505.8	4'879.7	7'064.7
Avg. Data Rate per Attacker (Mbps)	1 (all)	1 (all)	AN1: 2.49 AN2: 1.00 AN3: 0.69 AN4: 0.50 AN5: 0.30	AN1: 2.49 AN2: 0.99 AN3: 0.69 AN4: 0.50 AN5: 0.29
Avg. Packet Volume per Attacker (Pps)	2'974 (all)	1'300 (all)	AN1: 2'440.7 AN2: 975.0 AN3: 682.7 AN4: 488.4 AN5: 292.3	AN1: 1'219.9 AN2: 3'468.8 AN3: 683.4 AN4: 1302.3 AN5: 390.3

of different outcomes can be achieved regarding typical attack fingerprint characteristics, which includes packet size, data rate and packets per second.

Table 5.1 quantifies the difference in the composition of the four attack vectors in terms of packet sizes, data rates and packet volumes. The values for said table have been extracted using the python script found in Appendix D.3. The same observations as already made above based on the two figures can be made here too, though the table also reveals that there are slight variations in the data rate and packet volume metrics. For example, the average data rate, though configured to be equal for all four pulses, differs slightly. This can also be observed in the aforementioned figures, where slight fluctuations in the attack traffic can observed even within the same pulse. The cause for this is the configurable data rate fluctuation (*cf.* Section 4.3.4.1), which further contributes to making the generated attack traffic as flexible as possible.

This demonstrates the system's ability to generate pulses that differ in terms of their characteristics within the same overall pulse-wave attack.

VAR1 can also be used to demonstrate the configurability of two additional fingerprint metrics, namely source port and destination port numbers. Using a script to determine the port numbers (*cf.* Appendix D.2) as they are present within each attack vector leads to the port number distributions as seen in Table 5.2.

Both source and destination ports are configurable on a per-attacker and per-attack-

Table 5.2: VAR1: Port Number breakdown by Attack Vector

	Attack Vector 1	**Attack Vector 2**	**Attack Vector 3**	**Attack Vector 4**
Source Ports	Random	139 (39.99%) 185 (20.00%) 487 (19.99%) Random (20.02%)	not applicable	139 (22.79%) 185 (9.67%) 487 (18.43%) Random(49.11%)
Destination Ports	8080	Random	not applicable	9 (49.11%) 118 (5.52%) 777 (26.94%) Random (18.43%)

vector level. They both can either be randomized or set to specific values. Naturally, attack vector 3 which uses ICMP flooding does not make use of either port setting as it is not a transport layer protocol. TCP SYN flooding and UDP flooding however can fully exploit this to create diverse port number signatures. In attack vector 1 all packets share the same source port configuration (random) and are sent to destination port 8080 on the target. This is achieved through configuration of port settings on the attack vector, which has higher precendence and thus overrules the per-attacker configuration in this regard.

Attack vector 2 shows the result of not setting the the source port on the attack vector, which reveals the per-attacker settings. The fact that the percentages are not even splits is due to the data-rate fluctuation (*cf.* Section 4.3.4.2), which results in not every attacker having sent the exact same amount of packets. Attack vector 2 does however define the destination port as random for all packets.

In contrast, attack vector 4 does not prescribe any port numbers, hence the attacker-specific destination ports are now revealed. Attack vector 4 also illustrates the impact of packet volume on the percentages, as the previously even split in attack vector 2 is no longer present in the source ports of attack vector 4. This is due to the different PPS contributions of the individual attacker nodes as discussed earlier in this section.

These port numbers are not meant to be representative of an ideal configuration, but rather serve the purpose of demonstrating the ability of the system to create pulses whose attack vector composition also varies greatly in terms of the source and destination ports.

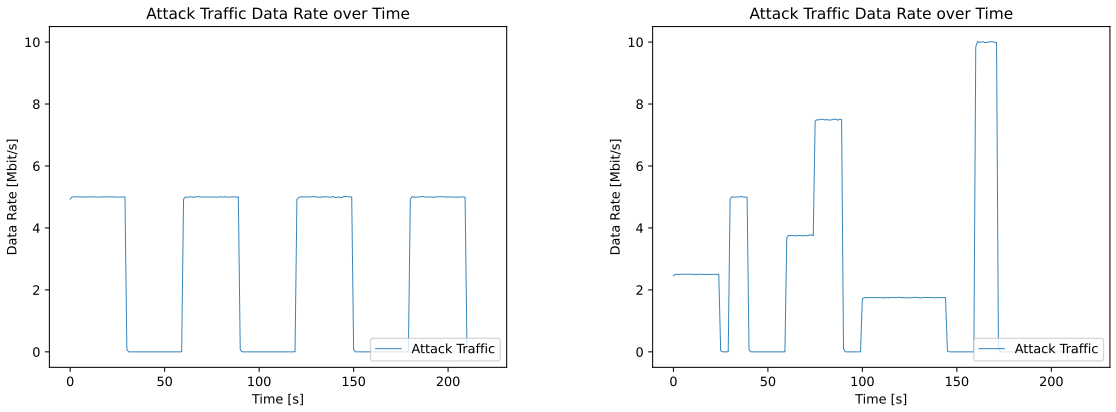
5.1.2 Variable Pulse-Wave Patterns

As demonstrated in Section 5.1.1, the system allows for a wide range of different options regarding the composition of attack vectors that on the surface look identical when simply considering their overall traffic pattern.

However, it is not a given, that all the pulses follow the same overall traffic pattern, the system also needs to support diversity in terms of the duration of the pulses, the length of

the time spent switching between targets, as well as being able to have pulses of different magnitude regarding the amount of traffic.

To illustrate the contrast, Figure 5.3a shows the pattern of the pulse-waves generated with scenario **VAR1**. As expected, the individual pulses do not differ from one another and match what was shown in Figure 5.1.



(a) VAR1: Constant Pattern (b) VAR2: Variable Pattern

Figure 5.3: Pulse Wave Patterns

The pulse-wave pattern generated by **VAR2**, shown in Figure 5.3a on the other hand is very different. Each vector uses a different data rate, pulse duration, and target switch duration. As is visible at the 75 second mark, the target switch duration can also be set to 0, resulting in a seamless transition from one pulse to the next. In fact, a switching duration of 0 is the default behaviour as otherwise the attacker nodes cease operation and resume them later, which goes against the principle of switching target on the fly as is done with pulse-wave attacks. The reason the switching duration can be configured is such that pulse-wave attacks can be simulated against a single target, with the switching duration mimicking the time the attack is directed somewhere else.

Both figures have been created with the script `plot_traffic_at_ixp_node.py`, which is available on the project's repository, as well as in the appendix (Appendix D.4).

It is important to note, that what is done with **VAR2** does not represent a typical configuration of the system, as in a pulse-wave attack usually more than one target is attacked (*cf.* Section 2.1.1.3) and there is likely no reason to artificially vary the data rates in such a way as whon in **VAR2**. However, for the sake of illustrating the configuration possibilities, **VAR2** only uses one target, otherwise, each pulse would occur twice in a row in the pulse-wave pattern.

As was shown, the system allows for the creation of a wide range of pulse-wave patterns, with the system doing little to restrict the user within the available configuration space.

5.2 Distributed Perspective

One of the main goals of the thesis is to be able to create datasets that represent a distributed view, *i.e.,* collecting data at different points in the CN topology and therefore being able to get a full view of traffic as it passes through a specific CN node. In order to demonstrate this distributed view, scenario **DIST** is introduced, which is specifically crafted to allow for the showcasing of specific aspects of the distributed perspective.

Table 5.3: DIST scenario

Scenario Name	**Number of CN Nodes**	**Number of Attacker Nodes**	**Number of Target Nodes**
DIST	8	12: - 5 in AS A1 - 3 in AS A2 - 2 in AS A3 - 2 in AS A4	2: - 1 in AS T1 - 1 in AS T2

Table 5.3 introduces the scenario with a brief list of its key characteristics. The configuration of the scenario is such that four ASs contain attacker nodes with two ASs each containing one target node. This leads to the topology as shown in Figure 5.4. The circles represent the individual CN nodes, with the red and green rectangular shapes representing the ASs configured in this scenario.

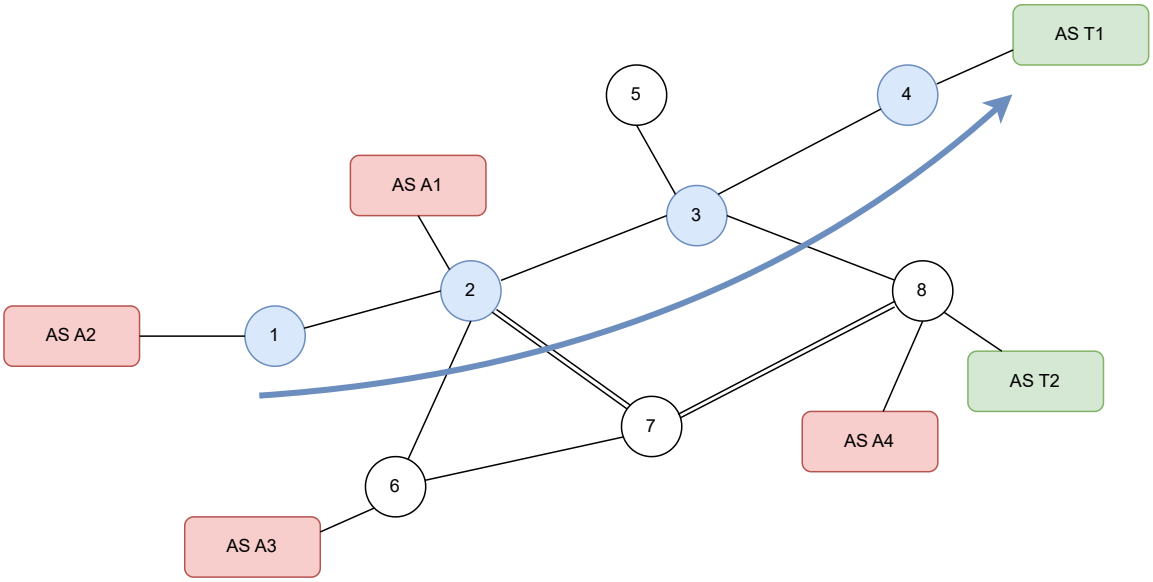
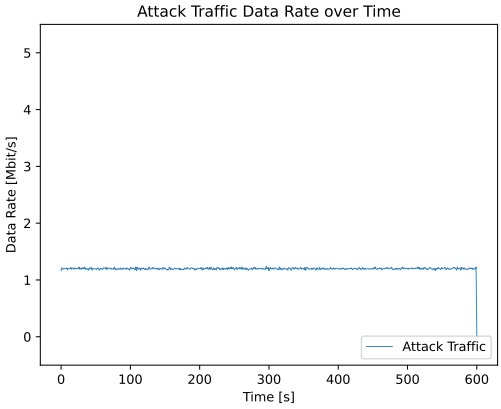


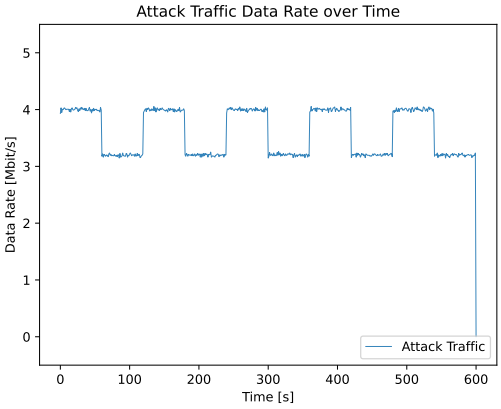
Figure 5.4: DIST Scenario Topology

In order to show how the different CN nodes have a different view on the attack traffic one specific path in the topology is chosen (along the blue colored CN nodes in Figure 5.4) and examined step by step. Only attack traffic as it leaves the CN node in question in

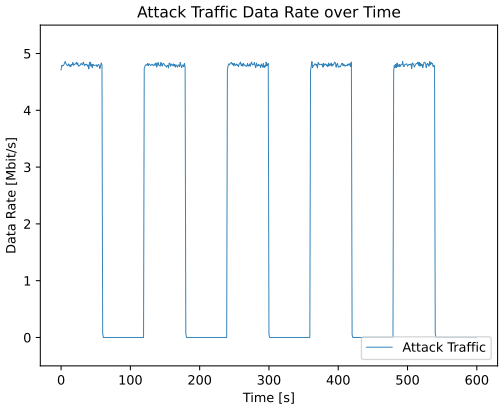
the direction of the target is shown. Given that this scenario merely serves the purpose of showing how the perspective on the attack traffic differs across CN nodes, the actual attack traffic is not configured to achieve high packet volumes or data rates, as will be evident when looking at the traffic plots representing the perspective of the four selected CN nodes.



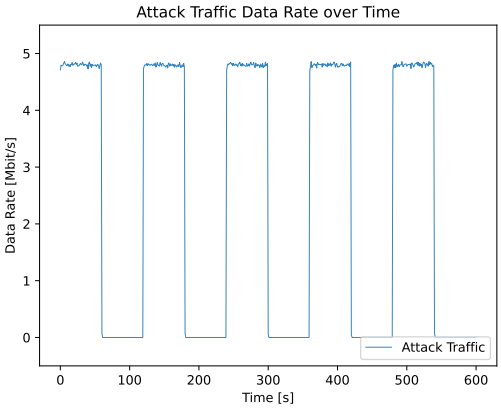
(a) DIST: Traffic at CN Node 1



(b) DIST: Traffic at CN Node 2



(c) DIST: Traffic at CN Node 3



(d) DIST: Traffic at CN Node 4

Figure 5.5: DIST: Traffic Pattern at different CN Nodes

Figure 5.5 shows the different views of the attack traffic along the path of the selected CN nodes. Starting with CN node 1 as shown in Figure 5.5a. At this point of the topology, the entire attack traffic of **AS A2** is shown, with the traffic pulses destined for both targets still appearing as a single block of traffic.

At CN node 2 (Figure 5.5b), the attack traffic of **AS A1** and **AS A2** both join the flow of attack traffic. Notably though, now a difference between the traffic for the first and second target is visible, and the uniform block of traffic is broken up. This is due to **AS A3** not contributing its attack traffic for target two because it has a shorter path when routing along the CN nodes 6, 7, and 8 towards **AS T2**.

Taking the perspective of CN node 3 as visible in Figure 5.5c the pulses meant for the two targets are now completely separate, due to all remaining attack traffic destined for **AS T2** breaking away towards CN node 8. At the same time though, the total amount of

attack traffic increases as the attack contribution of the nodes in **AS A4** join the flow of the DDoS attack.

Finally, the traffic as seen at CN node 4 is identical to the one at CN node 3, no further attack traffic joins along the examined path of nodes or is diverted from it. This then demonstrates the ability of the system to portray the traffic within the topology at different points, thus allowing for the examination of specific CN nodes' perspective which is a key goal of the system.

5.3 System Scalability

For the performance and scalability analysis a set of three differently scaled scenarios is used, as introduced in Table 5.4. The purpose of these three scenarios is to determine the performance cost when scaling up the simulation on a single machine.

The scenarios scale towards what a pulse-wave attack might look like if conducted within the SwissIX's topology, hence the limitation to 6 CN nodes, as the SwissIX operates with 6 peering locations [69]. Furthermore, all attacker nodes contribute equally in terms of traffic generation and the focus is put on having fewer nodes but with each of them producing traffic at a high data rate (though this naturally is limited by being ran on a single machine).

This goes back to what was discussed in Section 2.1.1.3, where the fact was stated that pulse-wave attacks tend to be seen as technically sophisticated, requiring precise orchestration and thus are likely best conducted with servers or virtual machines, rather than an IoT botnet, hence the reliance on fewer, but more high-capacity attacker nodes in these scenarios.

Finally, in terms of scenario design, the duration was set to 10 minutes, which is based on observations by [39] that most DDoS attacks today tend to run for shorter than 15 minutes. The average packet volume per second is calculated using the formula $packet\_volume = data\_rate/packet\_size$ ([13]) as already introduced in Section 5.1.1. Note that benign traffic was not considered and therfore the calculation only includes attack traffic.

For the purpose of enabling direct comparison at the same packet volume, the three attack vectors (UDP flooding, ICMP flooding, and TCP SYN flooding) were configured to use the same packet size. Given that SYN flooding uses fixed-sized packets, that packet size was thus applied to the other two vectors although they typically might use differently sized packets.

The number of targets remains locked to three, due to wanting all three configured attack vectors to have the opportunity to attack all targets. Increasing the number of targets would mean each attack vector has to perform one additional pulse, thus leading to an increased scenario duration. To ensure the number of targets is not a critical contributor to overall execution time it was also considered when testing for the performance impact of the individual factors (*cf.* Table 5.6).

Table 5.4: Scalability Scenarios

Scenario Name	**SC1**	**SC2**	**SC3**
CN Nodes	2	4	6
ASs	2	6	12
Attacker Nodes	5	15	30
Benign Nodes	10	20	60
Targets	3	3	3
Non-Target Servers	4	6	12
Approximate Average Packet Volume (Packets per Second) of Attack Traffic	14'800	66'900	178'500

The scalability evaluation scenarios were all ran on a MacBook Pro (Apple M2 Max with 32GB of RAM (random-access memory)), running Mac OS X Ventura version 13.3.1(a). To ensure the best possible basis for comparison between the different scenarios, the system was cut off from the internet during the runs, with no applications other than the prototype being active.

Unless stated otherwise, the prototype was granted full access to all 12 cores for MPI parallelization, and all simulations were conducted with the 'optimized' build profile available within the NS-3 framework.

Scalability was heavily limited by RAM consumption when initially starting the evaluation as the system began to leak memory quickly after scaling up minimally. Thus, an in-depth analysis was done, which in the end, revealed that the memory consumption issues were due to the MPI synchronization algorithm. Once the prototype was switched over to the alternative 'null message' based parallelization strategy, these memory issues disappeared completely (*cf.* Section 5.4.0.3).

Using the a script for capturing system resource consumption (*cf.* Appendix D.5) showed that both CPU and RAM remain locked at constant values throughout the simulation in all tested scalability scenarios, with NS-3 maxing out all cores granted to MPI, as long as they are assigned an actual task (*cf.* Section 4.3.1.1), and the system's combined memory consumption being at a constant value of a few hundred MB. Interestingly, the TCP SYN flooding vector shows slightly higher but nonetheless constant RAM consumption. Thus, for the scalability analysis, the focus is put on the time it takes the scenarios to finish, as that is the only way in which they differ in performance.

As can be gleaned from Table 5.5, where the resulting execution times for each scalability scenario are shown, the execution time scales only approximately proportionally with the (PPS) packets per second metric. From **SC1** to **SC2** the average PPS during active attack phases is increased by a factor of approximately 4.5, whilst execution time increases by a factor of 5.8. Similarly, when moving from **SC2** to **SC3** the PPS increases by a factor of 2.7 whilst the execution time is increased by a factor of 4.3.

Table 5.5: Scalability Scenario Results

Scenario Name	**SC1**	**SC2**	**SC3**
Approximate Average Attack Traffic Packet Volume (Packets per Second)	14'800	66'900	178'500
Execution Time	153s	887s	3'821s

This indicates that there are additional factors that impact execution time. This cannot be differentiated based on the three basic scalability scenarios. Hence, variations of **SC2** are introduced, which increase one individual factor (*e.g.,* the number of benign nodes) to the value used in **SC3** while keeping the rest of the configuration at the values used in **SC2**.

Table 5.6: Performance Impact of Individual Factors

Configuration Name	**Configuration Difference Compared to SC2**	**Execution Duration**	**Excecution Duration Change Compared to SC2**
SC2	✗	887s	✗
SC2_AN	Doubles the Number of Attacker Nodes. Halves the per Attacker Data Rate to ensure same PPS as SC2	851s	- 4.1%
SC2_AS	Doubles the Number of ASs	1'112s	25.4%
SC2_BN	Doubles the Number of Benign Nodes	859s	- 3.2%
SC2_CN	2 more IXP Nodes (1.5 times incrase over **SC2**)	909s	2.5%
SC2_NT	Doubles the Number of Non-Target Server Nodes	888s	0.1%
SC2_PV	Same PPS as SC3 (2.7 times increase over **SC2**)	2'790s	314.5%

Looking at the impact the individual scaling factors have on the execution time as shown in Table 5.6, a few remarkable results jump out. It is clear that by far the most impact on the execution time is caused by increasing the **packet volume** through the PPS metric as is done with **SC2_PV**. This is not surprising, as creating and modeling the routing and processing of additional packets can easily be conceived as computationally expensive.

The second most impactful factor appears the be the addition of additional ASs in **SC2_AS**, which could make sense as additional ASs increase the size of the overall topol-

ogy potentially by a significant amount, depending on the nodes present within them. However, the total number of nodes in the topology (attackers, targets, benign nodes, non-target servers) has not changed, they have simply been re-distributed onto a larger number of ASs. As such, the cause for the increase is in execution time is unclear.

What is surprising is the fact that there are a number of factors that consistently **reduce** the execution time when increase, even though the reductions are small. **AN_BN** shows consistently lower execution times than **SC2** which is even more suprising given that more benign nodes being present means that more benign traffic and therefore more packets are being modeled.

SC_AN suggests that having a larger set of attacker nodes whilst operating at the same combined packet volume is beneficial for the execution time. What causes this is unclear, and it could also be that this effect ceases once a certain scale is reached.

To test this, **SC3_120AN**, a varitation of **SC3** is introduced that keeps all factors the same but quadruples the number of attacker nodes, increasing their count from 30 to 120, whilst keeping the overall packet volume at the same level by reducing the per-attacker data rate by a factor of 4. The results reveal that **SC3_120AN** is **more** expensive than the regular **SC3** in terms of runtime, requiring 4'725 seconds to complete, which marks a 23.7 % increase over the regular **SC3**.

This suggests that there are more intricate interactions at play between the different factors and that their impact is not necessarily consistent as the scale of the use cases increase. This does raise some concerns about the higher-end scalability of the system as potentially more and more factors could start having an increased impact on the execution duration.

However, determining what that interplay between factors looks like at higher scales requires a more dedicated and more time-intensive analysis and lies outside of the scope of the thesis.

As for increasing the number of targets: Though not shown in Table 5.6, altering the number of targets has no impact on performance, unless attack vectors are configured such that they exhibit different performance (*e.g.,* having unequal packet per second stats) and the duration of the use case is configured such that not all vectors can attack all targets the same amount of time. This in turn then introduces a situation where the different execution time costs receive different weights depending on how many times they can complete within the simulation run.

5.4 Takeaways from Developing with NS-3

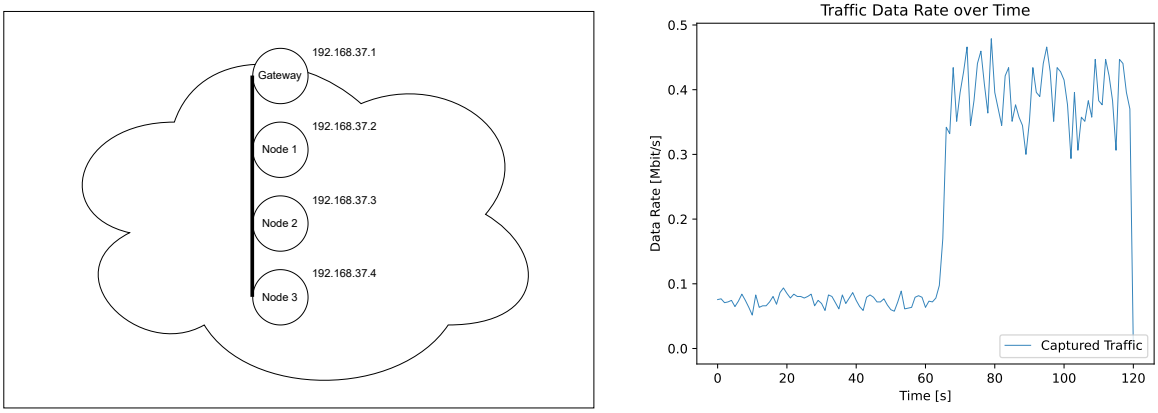
Starting the evaluation phase quickly revealed a number of severe problems that needed fixed for the system to pass any reasonable scrutiny regarding its ability to generate pulse-wave patterns reliably.

Because these issues had to be addressed during this thesis and were not aspects that could be delegated to future work, they have been fixed and thus do not affect the final

evaluation as laid out in Sections 5.1, 5.2 and 5.3. Regardless, they are findings that relate to performing system evaluation and they provide important takeaways about working with NS-3, hence the decision was made to include this section about findings during what one might call a 'preliminary evaluation phase'.

5.4.0.1 Unreliability of the CSMA Channel

As was briefly hinted at in the implementation part of the report when discussing the AS (*cf.* Section 4.3.3.2), the initial AS implementation relied on a CSMA Channel, which offered a convenient way to create the internal topology of the ASs, include all the necessary nodes and perform address assignment in a straight forward manner.



(a) CSMA AS Internal Topology (b) CSMA Scenario Traffic

Figure 5.6: CSMA Channel Issues

Figure 5.6a shows what such a CSMA-based AS looks like, with the internal nodes essentially forming a bus-topology starting from the AS gateway.

Whilst this is a straight-forward though simplistic approach to implementing an AS, it suffers from the fact that apparently on CSMA channels a large part of the packets does not make it past the gateway. What exactly happens to them is unclear, but the results clearly indicate that the CSMA channel is not suitable for this particular purpose.

Figure 5.6b makes this very apparent, as it shows the attack traffic as captured at the exit of the attacker nodes' AS. One can scarcely make out the hint of three pulses, starting at the 60 second mark. What is lost completely is the first three pulses, which are supposed to be visible in the first minute of the run.

This illustrates the issue rather well, as based on what the author was able to observe the CSMA channel appears to struggle more as the packet volume increases, which is the case within the first minute, where pulses with rather small packet sizes (and thus higher packet volume per second) are situated.

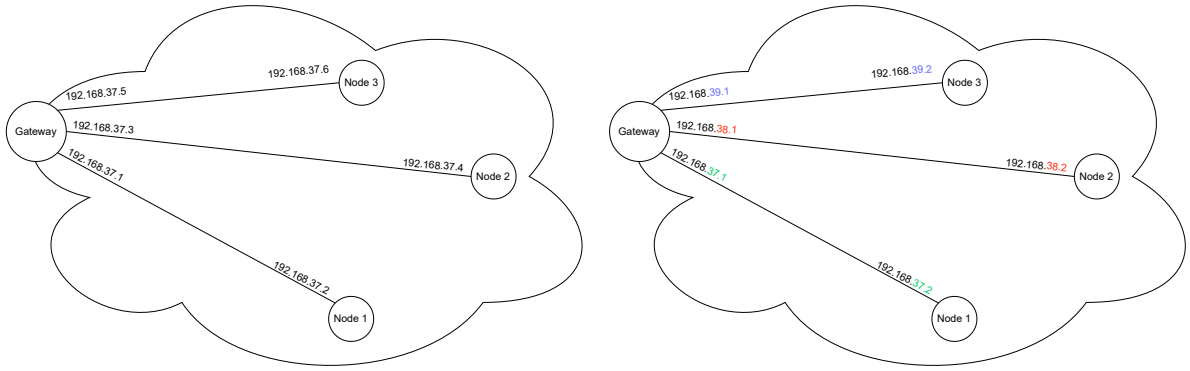
Another clue that something was not working as intended would have been the file sizes of the PCAP output, which for the CSMA scenario are approximately 6 MB, which does not seem appropriate for the configured 2 minutes of attack traffic at multiple Mbit/s.

The observation that the CSMA channel suffers from packets being lost is not limited to just this prototype. A minimal topology (*cf.* Appendix D.7) of just two nodes, using none of this system's custom classes shows the discrepancy of using the CSMA channel when compared to using a point-to-point channel. When running that minimal topology with the CSMA channel, the resulting file size is 6.7 MB, with a packet count of 7'634, whereas the same topology when ran with the point-to-point channel produces a PCAP file of 696 MB, containing 1'190'185 packets. In that particular case, the CSMA channel seemingly has only transferred 0.6% of all packets.

The author wishes to make clear that this is **not** to be taken as an assault on neither the capabilities of NS-3 nor their contributors. The CSMA channel may simply not have been intended to be used in such scenarios. Regardless, for this thesis this meant having to rethink and re-implement the way the ASs construct their internal topology.

5.4.0.2 Time-to-Live Issues with Single Subnet AS

The next step in getting the AS to perform properly was switching to a point-to-point channel-based internal AS topology and using a single subnet to assign addresses to the resulting interfaces. Figure 5.7a shows how the resulting topology and address assignment scheme operates.



(a) Single Subnet Address Assignment (b) Per-Channel Subnet Address Assignment

Figure 5.7: Point-to-Point AS Address Assignment Schemes

The issue with the single subnet approach to address assignment was that it led to what the author suspects are ultimately routing issues. These express themselves in most packets exceeding their Time-to-Live (TTL) and never reaching their destination. The discrepancy in the size of the PCAP files capture at the exit of the attackers' AS and the entry of the targets' AS illustrates this well. Whilst one would expect both files to be equal in terms of the captured packets (at least in this scenario, where only two ASs are configured and the attackers and targets are separated into one AS each), the file at the exit of the attacker nodes' AS is 596.4 MB large. In contrast, the one at the entry to the target nodes' AS takes up only 56.8 MB. In terms of packet counts, this equates to 7'633'852 and 341'820.

What is remarkable about this issue is that this single subnet approach to assigning addresses to a topology of point-to-point channels was used **without issue** in the CN. The author **suspects** that this address assignment scheme only becomes an issue once the corresponding nodes are used as source or target for packets rather than just nodes that route traffic, as is the case with CN nodes. This is merely speculation, and the author could not produce any sources or information to back that hypothesis.

Listing 5.1: Single Subnet Address Assignment Approach

```
1  NetDeviceContainer mergedDeviceContainer;
2  PointToPointHelper p2p;
3  Ipv4AddressHelper address;
4  address.SetBase(addressBase, addressMask);
5
6  for (int i = 1; i < m_numNodes; i++)
7  // excluding 0, since that is the index of the gateway node
8  {
9          NetDeviceContainer devices = p2p.Install(nodes.Get(0), nodes.Get(i));
10         // continuously merge devices of new channels
11         mergedDeviceContainer.Add(devices);
12 }
13
14 // assign addresses all at once on same subnet
15 Ipv4InterfaceContainer interfaces = address.Assign(mergedDeviceContainer);
```

To briefly demonstrate the single subnet approach, its relevant parts are shown in Listing 5.1. Instead of assigning addresses on each individual `NetDeviceContainer` that results from setting up a point-to-point channel between two nodes (line 9), a separate `NetDeviceContainer` is instantiated outside of the loop (line 1) and any new `NetDeviceContainer` instances are continuously merged into it (line 11). This then enables the address assignment to be done all at once, without the need for manual management of subnets or address iterations (line 15).

Ultimately, the single subnet approach had to be abandoned in favor of using a separate subnet for each individual point-to-point channel, as shown in Figure 5.7b and explained in more detail in Section 4.3.3.2.

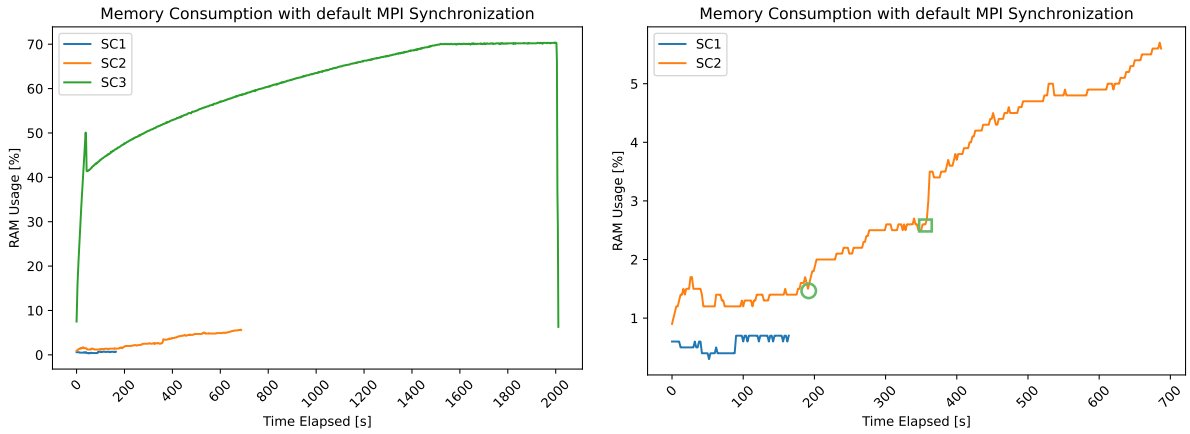
Since the exact cause of the TTL issues was not identifiable, the decision was also made to no longer use the single subnet approach when performing address assignment on the CN, despite it having worked without issue there as there may be other side-effects that are less visible.

5.4.0.3 Memory Leak with Default MPI Synchronization Algorithm

When initially conducting the scalability analysis, the prototype was leaking memory heavily. Figure 5.8a shows the memory consumption of the three scenarios, with Figure 5.8b showing the same data but without **SC3** to provide a more detailed view of **SC1** and **SC2**.

Whilst **SC1** and **SC2** do run fine in terms of execution duration, the pattern of continuously increasing memory that starts with **SC2** effectively imposes a scalability ceiling on the

system, as RAM will eventually run out when the scaling of the scenario reaches a certain threshold. This is shown with **SC3** where the system's memory consumption escalates to approximately 70% which amounts to approximately 84% when not subtracting the RAM consumption of the idling machine. At 84% RAM usage the system is forced to start using swap memory ([50]) and the simulation progress slows down significantly until the system crashes.



(a) SC1 to SC3: Memory Consumption (b) SC1 and SC2: Memory Consumption

Figure 5.8: Memory Consumption when using default MPI Synchronization

This led to a lengthy process of investigating the memory consumption behaviour of the system, which did yield some notable insights. The main factor turned out to be the packet volume, regardless of packet size and data rate. This indicates that there is some relationship tied to the amount of sent packets and memory consumption. A report by Yoav Levy in an NS-3 community forum post from earlier this year [49] shows that there is an issue where scheduled events stay in memory throughout the entire simulation run depending on how they are handled.

Given the way data rate is implemented in the `OnOffRetargetApplication` which performs the attack traffic generation, the system creates one send event per sent packet, which made this a promising lead.

However, when looking at the per-process RAM usage with *e.g.,* the 'Activity Monitor' tool ([7]) and mapping the processes to the MPI ranks for which they are responsible reveals that this is unlikely. As shown in Table 5.7, the main bulk of the RAM is used by the processes responsible for MPI rank 0 and rank 2 (*cf.* Section 4.3.1.1).

Table 5.7: Memory Consumption Breakdown by MPI Rank

MPI Rank	**Responsibility**	**Percent of Total Memory Consumption**
0	Central Network	28.2%
1, 3, 4, 5 and 6	AS 1 (Attack Traffic Generation)	$< 1\%$ combined
2	AS 2 (Target Nodes)	71.3%

This does not support the theory that the RAM usage is tied to scheduled send event, as those should be handled by all MPI ranks **except** rank 0 and rank 2, as mpi ranks 1 as well as 3 through 6 are tasked with attack traffic generation in the scenario from which these consumption figures were taken.

Another observation is that, as indicated on the RAM usage curve on **SC2** in Figure 5.8b, when the attack vector changes over from UDP flooding to ICMP flooding (highlighted with a green circle) and when changing to TCP SYN flooding (indicated by a green square) the RAM consumption experiences a significant increase. This was seen as being due either the different underlying protocols of the attack vectors or due to the fact that ICMP flooding and TCP SYN flooding use the raw socket implementation to send their packets.

Note, that since doing this analysis the UDP flooding vector has also been switched over to the raw socket due to allowing for increased configuration, thus that difference in performance may no longer hold.

Interestingly, this difference disappears as the overall packet volume increases. In Figure 5.9 **SC2_HPV**, a variant of **SC2** with increased packet volume shows, the characteristic memory consumption curve that kicks in, leaping to approximately 40%. Unless the packet volume is too high for the system, as is the case with **SC3** the system prefers to stabilize at around 40% where the difference in RAM usage between the different attack vector implementations disappears.

Further investigation showed that installing a so-called `PacketSink` application on all target nodes lifts the scalability ceiling. According to the NS-3 documentation, a packet sink is tasked with receiving and consuming incoming packets [57]. The way the attack vectors are currently implemented they simply send their packets to the target nodes without any dedicated applications being present on the target nodes to consume the traffic. This hints at packets still taking up memory after arriving at their destinations, thus also explaining why rank 2, the MPI rank responsible for the target nodes in these scalability scenarios, shows escalating memory consumption.

SC2_HPV_PS in Figure 5.9 shows the difference a UDP packet sink makes when present on all target nodes, with memory consumption being markedly lower during the UDP flooding part of the scenario until the simulation switches to an attack vector that is not UDP based as indicated with a green circle on the plot.

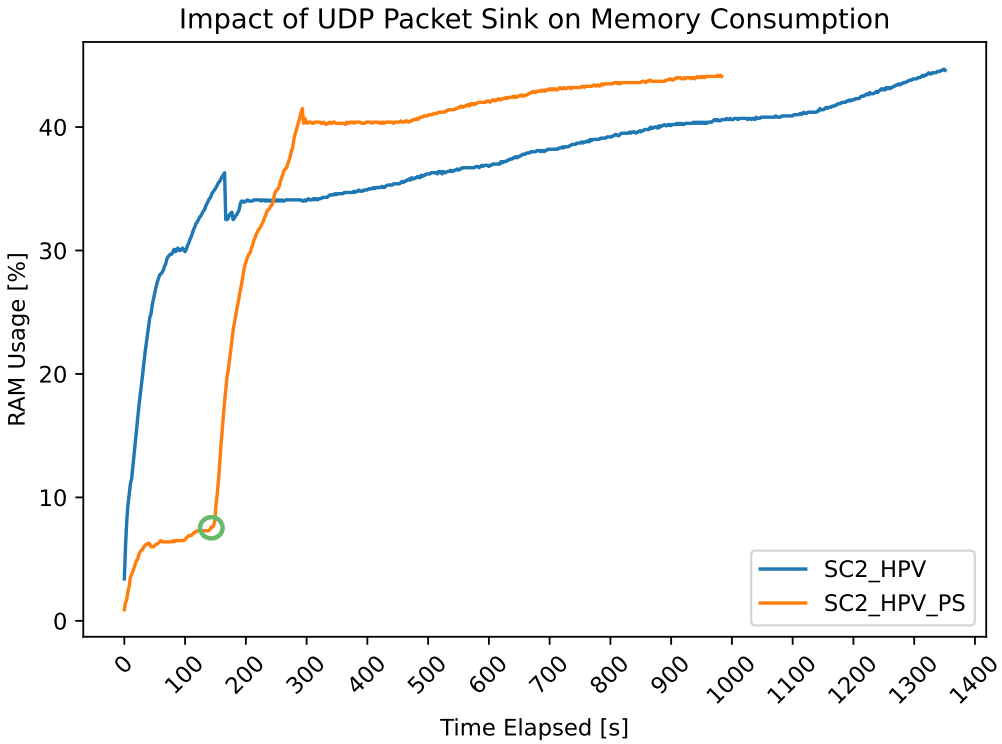


Figure 5.9: Impact of UDP Packet Sink on Memory Consumption

However, whilst installing a packet sink for UDP packets is simple, doing so for ICMP, in particular, turned out to not be possible within the time frame of the thesis. On top of that, even if packet sinks were available for all the required attack vectors, they would only solve the memory consumption problems on MPI rank 2, with rank 0 still being unchanged, thus remaining a scalability limiting factor.

Ultimately, the MPI rank 0 RAM consumption issue turned out to be MPI-related. The NS-3 documentation lists two possible simulator implementations that can be used for MPI [55]. The default implementation relies on a global parallel synchronization strategy, whilst the alternative 'null message synchronization algorithm' does not [55].

As the documentation states, which of the two synchronization strategies should be used depends on the scenario, one tries to model [55]. Generally speaking, the null message synchronization approach scales better, except in scenarios where there are long periods without events [55]. Given the nature of pulse-wave attacks, this may occur, as when targets are distributed to different ASs, there could be MPI ranks that do not receive any traffic nor scheduled events. This should, in principle, support the decision to stick with the default synchronization algorithm.

However, the fact that MPI rank 0 showed such high memory usage led the author to suspect that this global coordination of the default synchronization strategy may be at least a part of the problem. Especially since rank 0 does (at least in **SC1** and **SC2**) only focus on the CN. The CN itself could potentially cause high RAM usage, as it does trace packets to create the PCAP output files. However, turning off tracing did not affect memory usage. As it turns out, the suspicion that the memory leak was tied to MPI

turned out to be correct, as switching over to the null message-based synchronization **completely** solved all memory problems.

All the observations above about the leaps in memory use when attack vectors change, the impact of packet volume in general, and the increased scalability when using packet sinks were specific to the default synchronization algorithm. Once the switch to the null message synchronization algorithm was made, memory usage remained low and consistent no matter the scale of the scenario, as is shown in Section 5.3 where system scalability is discussed.

5.5 Discussion

The thesis's main goal is to generate distributed datasets specifically for pulse-wave DDoS attacks. Given that there are no existing pulse-wave data sets to compare against the datasets generated with this prototype, it has to be evaluated in isolation. Nonetheless, criteria for successfully meeting the main goal can be established and evaluated, specifically with regard to how select key metrics that relate to attack fingerprinting are expressed in the datasets and if they match the configuration input.

This is done in Section 5.1 where the system's capabilities regarding the generation of **diverse** traffic attack traffic patterns as well as the ability to compose a wide range of **differently structured** pulses in terms of configurable attributes such as protocol, packet size, packet volume, port numbers, and data rate are demonstrated. The system also allows for many of the attributes to be configured on a global level, a per attacker node level, or a per attack vector level (*cf.* Section 4.3.2.2). This enables the creation of pulse-wave DDoS attacks with unequal contributions across attacker nodes, extending the space of possible use cases the prototype can model.

For example, a configuration may be used where the attacker nodes all contribute equally to the attack, thus representing a use case that *likely* has attackers use servers or virtual machines to perform the attack, resulting in precise orchestration and resource use. Alternatively the individual attacker nodes could be configured to have greatly different data rates and packet sizes, thus resulting in very different request per second numbers per attacker, which *may* be more likely to match the results of an IoT botnet based attack.

The prototype is capable of creating different pulse-wave patterns in terms of the overall duration of the attacks, the number of attack vectors used, and the duration of the pulses as demonstrated in Section 5.1.2.

A further criterion that is used in attack fingerprints is TTL (time-to-live) which is indicative of the number of hops taken by a packet on its way towards the target (*cf.* Section 2.1.1.5). Although not explicitly evaluated, the TTL figures of attacks simulated by the system are determined by the topology of the configured use case and therefore also indirectly configurable. Along the same lines and also not explicitly evaluated, IP addresses of attacker nodes are not directly configurable but are steered indirectly by controlling in which AS how many attacker nodes are present with the IP address space of ASs being configurable.

AS numbers are in principle subject to the user's configuration input as the IP address space of individual ASs can be specified. Therefore a fingerprint creation tool such as the one discussed in [34] may try to derive an AS number based on the source IP addresses of the attack traffic. It must however be said that the prototype does not attempt to delineate the IP address spaces of different ASs based on the address spaces of real-world ASs. Thus, fingerprinting tools may struggle to do so which marks an area of potential improvement in the future. Another point that must be made is that due to the late changes to the AS that were necessitated as explained in Section 5.4.0.1, the AS are quite wasteful in terms of how they use their available IP address space, which may require future work to investigate different options for address assignments.

As for the **distributed perspective**, the prototype captures traffic at all interfaces of the CN topology, thus enabling a complete dissection of the traffic on all CN nodes of the topology. A demonstration of how the attack traffic pattern changes as the perspective of different CN nodes is taken is given in Section 5.2. This satisfies the criteria of creating a distributed dataset. Furthermore, the configuration allows for the creation of a wide range of use cases, with topologies of varying sizes, different delays, and different numbers of ASs and nodes, thus providing flexibility in terms of the distributed scenario that is to be simulated.

As such, the main goal of the thesis is met, although there is room for refinement in certain regards. For example, the generated attack traffic **within** a given pulse remains relatively static across the pulse duration. Much can be configured to create very different pulses, but any given pulse does not show much dynamicity. Whilst the prototype does *e.g.,* model fluctuations in data rates for each attacker node (*cf.* Section 4.3.4.2), these fluctuations are local to each node and, as such, do not lead to a very noticeable impact on the attack traffic pattern. This could be supplemented in the future with *e.g.,* modifications to the routing code of NS-3 to simulate packets being dropped or model general unreliability factors such as attacker nodes failing and quitting the attack or a specific connection on the topology suffering from varying throughput and thereby affecting groups of nodes. This then would lead to more impactful and ultimately realistic fluctuations in the attack traffic.

Another area of improvement is the internal modeling of ASs, which is rather pragmatic and simplistic in nature due to the time constraints of the thesis. As such the AS modelling could be refined such that ASs can be configured to use a range of different and more sophisticated internal topology models. Along the same lines, the approach taken with the simulator of providing the ability to create generic IXP topologies and attach ASs is limited in the sense that the resulting topology does not explicitly implement the traditional three-tiers of ISPs ([17]).Therefore the simulated topology simply consists of IXPs and ASs, with the same AS model being used to simulate any AS no matter what it represents semantically. Ultimately an AS, as implemented within the prototype represents a specific range of IP addresses used to grant addresses to nodes within them, which may be too generic for certain use cases.

An aspect that has not been mentioned yet is the choice of a benign traffic model. As outlined in Section 4.3.4 the benign traffic application is an existing implementation of a model introduced by [62] in 2012 with a focus on HTTP traffic. As such, it is well-suited

to model benign traffic for some use cases and less well-suited for others. Implementing additional models for simulating other types of legitimate traffic and offering a selection of models to choose from would thus increase the quality of the generated data sets for cases where benign traffic in the form of an HTTP-based model is less fitting.

The system does fulfill the design requirements established in Section 3.1. Requirement **R1**, **R2** and **R3** capture aspects of the main goal of the thesis *i.e.,* the distributed perspective and the configurability of diverse pulse-wave patterns, respectively. These requirements are met, as already discussed earlier in this section.

The system's richness in configuration possibilities comes at a price in terms of ease of configuration. Whilst the project's repository contains detailed explanations regarding the individual configuration parameters and an example configuration, the configuration still requires some effort on the user's part. This is less of an issue when a given user has an exact idea about the use case they wish to configure. However, when wanting to run the system without caring too much about configuring individual nodes or ASs, the user may prefer a convenient way to configure more generic use cases. The system currently does not support this well and should be improved in that regard in the future.

When it comes to **R4** which is centered around reproducibility, the system can satisfy that requirement. Use cases and specific scenarios are controlled by the configuration file which can easily be shared, allowing for the reproduction and verification of results. Given the open source and purely simulation-based nature of the prototype, the code is available to anyone, and setting the project up also does not impose any special hardware requirements. It simply requires a computer that can run NS-3 , thus achieving high reproducibility and allowing anyone who wishes to extend the system to do so.

This is where requirement **R5** comes into play. **R5** is focused on extendability, which is a key requirement for a prototype such as this. The system is fundamentally a proof of concept that lays the foundation for future enhancements with some areas for potential future work already highlighted throughout this section of the report. The prototype is implemented as an NS-3 module, thus it can in principle be included in any existing NS-3 installation and be used in conjunction with other modules.

The prototype follows a modular approach, using a set of different class hierarchies to cover different aspects of the system such as configuration parsing, topology building, or node behaviours. This makes the prototype open for extension and straight-forward to build upon.

Extending the system does however require a good grasp of NS-3 , which can be challenging to work with not only because of the sheer amount of possibilities the framework provides but also due to its intricacies and sometimes unexpected behaviors as also experienced throughout the work done on this thesis (*cf.* Section 5.4).

Scalability is difficult to judge. Whilst the scalability analysis done in Section 5.3 is generally favorable, with the main governing factor of the execution duration being the amount of simulated packets, there are some questions about how the impact of other factors, particularly the number of attacker nodes, impacts the duration at higher scales.

As such the thesis cannot definitively state how the execution time will develop as individual factors are increased. A more in-depth analysis could be conducted in the future to attempt to disentangle the performance impact of individual configuration parameters more clearly.

What can be said is that the system runs stable, with low and consistent memory usage whilst making full use of the cores the simulation is allowed to use, provided that the correct parallelization synchronization algorithm is used *cf.* Section 5.4.0.3.

The prototype thus provides **novelty** in the types of datasets it generates, focusing on creating distributed pulse-wave datasets that provide a more holistic view of an attack compared to the traditional single-view datasets and thereby contributing to the development of collaborative DDoS mitigation approaches.

Chapter 6

Final Considerations

6.1 Summary

In this thesis, a literature review centered around the topics of DDoS dataset generation techniques and pulse-wave DDoS attacks was conducted. This laid both the theoretical foundation of the subsequent design and implementation activities but also served as starting point for discussing related work relating to the aforementioned topics. The review of related work both served as an analysis of existing work in this domain as well as a means to define the research gap and thereby justify the thesis.

Following the selection of the traffic generation framework, NS-3, the prototype design was developed based on requirements that have been derived from the thesis' goals prior to that. The decision was made to pursue a generic design regarding the topology that is made available by the prototype, *i.e.,* not implementing a specific use case but rather providing a system that can model a wide range of use cases based on configuration input, with the generic structure of the use cases being given by a central network made up of IXP nodes and ASs that can be individually defined and attached to the IXP node of choice. This was done to provide high flexibility in terms of the attack scenarios and topologies that can be simulated. This is crucial, due to no public pulse-wave DDoS datasets being available and therefore no statistical analysis could be conducted that might inform design decisions regarding what type of topology is best suited to create pulse-wave DDoS datasets.

The same approach was also taken with regard to implementing the configurability of the attack vectors with regard to common DDoS fingerprint properties, putting the focus in maximizing flexibility and allowing for the maximal range of possible outcomes, rather than trying to replicate existing attack datasets of non-pulse-wave attacks (given that no pulse-wave attack data is publicly available).

The resulting prototype was evaluated with regard to its ability to produce different pulse-wave traffic patterns, simulate pulse-wave DDoS attacks, and provide the required distributed view onto the flow of traffic within the simulated topology. Further, the scalability of the system was evaluated. The evaluation results were discussed and contrasted

with the goals of the thesis as well as the design requirements and shortcomings were mentioned which provide starting points for future work.

6.2 Conclusions

Overall, the thesis manages to achieve its goals. The prototype is capable of generating a wide range of different pulse-wave patterns using different attack vectors, durations, and composition (*cf.* Section 5.1). The generated dataset, produced in the form of PCAP files, fulfills the requirement of being distributed, *i.e.,* providing a view of an attack from different points within the topology of the simulated scenario (*cf.* Section 5.1.2) and exhibiting the desired characteristics with regard to properties commonly used in the realm DDoS fingerprinting.

This is achieved by exposing a large suite of parameters to users in the configuration file (*cf.* Section 4.3.2.2) which allows for the flexible creation of diverse use cases by specifying the desired topology in terms of IXP nodes, ASs which can be populated with different types of nodes. The pulse-wave traffic patterns are configurable in terms of duration, with a suite of properties such as source and target port numbers, attack vector (protocol), packet sizes and data rates also being available for configuration.

Thanks to its approach which prioritizes high flexibility in the kinds of use cases that can be configured, the prototype is not bound to any specific existing datasets as it does not use their statistical properties (*e.g.,* packet size or average time between packets) in its attack traffic model but rather leaves all of that up for specification by the user through the configuration file. This represents both a strength as well as a potential weakness, as on the flip-side the generated traffic may be too generic for certain more specific use cases. Due to the proposed system's prototype nature, and the time constraints of operating within the scope of a thesis, not every aspect of attack traffic is configurable in as much detail as might be desirable.

An concrete example of this is the modelling of the fluctuations in the attack data rate or in other words the variation in time between packets. As discussed at the end of Section 4.3.4.2, the prototype uses a uniform distribution with a configurable range to calculate that fluctuation. However, it may be that for certain use cases different, less even distributions are more appropriate, which is something the prototype in its current version does not support. In absence of publicly available pulse-wave datasets to derive a more nuanced distribution from, an option that was considered is to base the modelling of data rate fluctuation on non-pulse-wave datasets. However, these attacks exhibit different properties, such as *e.g.,* the pronounced ramp up (*cf.* Section 2.1.1.3), which would not align with the structure of a pulse-wave attack and thus might produce an adverse effect on the quality of the produced datasets. This highlights the importance of having publicly available pulse-wave datasets (or datsets in general).

Performance is governed primarily by the overall packet volume, as discussed in in Section 5.3. Memory consumption is no concern, given the right choice of synchroniziation

algorithm for parallelization and therefore the scalability is only limited by the amount of time the simulation is allowed to run for.

As such the prototype is projected to be be able to simulate larger scale use cases if given enough time to run, though the scalability analysis has not been able to entirely clear up the impact of the individual scaling parameters, with especially the impact of the number of attacker nodes raising some questions about higher-end scalability.

The thesis also provides a number of insights specifically with regard to challenges faced in the context of working with the NS-3 framework. Whilst it is undoubtedly a powerful framework, it is in the author's opinion difficult to use at times. This has manifested itself in a number of major challenges in the later stages of the thesis discussed in detail in Section 5.4. Of those the biggest takeway is the importance of selecting the right synchronization algorithm for parallelization as the default algorithm lead to significant the memory consumption problems. Analyzing and consequently changing parts of the system to overcome those challenges put considerable strain on the timeline, though doing so was necessary as otherwise the prototype would have been rendered essentially non-viable.

A further aspect that bears mentioning is that during the selection of the traffic generation framework, a broader decision had to be made regarding the direction of the thesis. Specifically, whether the prototype should use emulation or simulation as the technique of choice. Initially, the thesis was meant to focus on emulation, but when evaluating possible frameworks and discussing the findings with the supervisor the decision was made to instead build a simulator. The main reason for that choice was the inherent scalability limitations of emulators (*cf.* Section 2.1.2.3).

Overall, the contribution made by this thesis represents a significant step towards having accessible and diverse distributed pulse-wave DDoS datasets, which can serve as a basis for further research in the realm of pulse-wave DDoS attacks and inform the future development of cooperative DDoS defense mechanisms.

To that end, the prototype is made publicly available, fostering reproducibility and allowing for the prototype to be expanded upon in the future. Additionally, the prototype employs a modular, object-oriented architecture that allows for the straight-forward extension and enhancement of its capabilities. As such the requirements of reproducibility and extendibility put forth in Section 3.1 are fulfilled.

6.3 Future Work

The prototype system proposed by this thesis marks a strong first step towards the generation of distributed pulse-wave DDoS datasets. However, as discussed in Section 5.5, there are parts of the system that could be improved upon in the future.

A number of models used in the system are limited in terms of their realism. Due to the limited scope of the thesis, the AS implementation is relatively simplistic and cannot model more complex internal topologies, such as nested subnets. Depending on the use

case the user wants to configure, having this capability would be desirable, increasing the realism of the topology.

Another model that could be improved in terms of realism concerns the attack traffic generation. Whilst the prototype is strong at generating pulses with very different internal makeup in terms of factors such as protocols, packet sizes and packet volume, this could be expanded upon. For example, attacker nodes cannot change the size of their packets throughout a given pulse and the prototype does not offer the possibility to have attacker nodes join are drop from the attack. Additionally, whilst the data rates of the individual attacker nodes does is subject to a configurable degree of variation throughout the attack, more global effects that have a more noticeable impact on the overall traffic patterns could also be desirable, such as for example having groups of nodes experience reduced data rates or dropped packets due to a connection in the topology suffering from reduced throughput. Effects such as this would lead to overall less consistent attack traffic and could improve the realism of the generated attack traffic pattern, though this certainly would require further research into both what types of effects are appropriate and what can be achieved within the confines of the NS-3 framework.

In terms of ease of configuration there are also improvements that could be made. The high degree of configurability of the system results in having to dedicate some amount of effort to configure a specific use case. Whilst the individual parameters are well-explained, the system could still benefit from providing a more streamlined configuration experience.

A different avenue of possible enhancements is adding additional attack vector implementations and benign traffic models to further expand the range of configurable use cases. Furthermore, the attack traffic generation may benefit from having additional options to chose from in terms of how the data rate fluctuations are modelled.

The list of typical fingerprint metrics that are made configurable in this prototype is also not exhaustive. For example, another criterion that is considered in the realm of DDoS fingerprint creation is whether or not IP addresses have been spoofed ([34]). This could prove challenging to achieve within the confines of NS-3 and require a higher degree of technical understanding of the framework, but would further increase the quality of the generated datasets.

Another area of potential improvement or at least further investigation is the scalability of the system. Whilst the system runs stable, the fact depending on the overall scale of the scenario that the number of attacker nodes in particular can have an either positive or negative impact on the execution duration indicates that scalability could be worth revisiting in future iterations of the prototype.

The list of potential future work discussed above is certainly not exhaustive. Ultimately, as the prototype was specifically built such that future enhancements are possible and provides a strong foundation to expand upon, regardless of which avenue future development is ultimately considered, be that the refinement of already existing models, the addition of entirely new capabilities, or more performance-centric improvements.

Bibliography

[1] Yahya Al-Hadhrami and Farookh Khadeer Hussain. "Real time dataset generation framework for intrusion detection systems in IoT". In: *Future Generation Computer Systems* 108 (2020), pp. 414–423.

[2] Mouhammd Al-Kasassbeh, Ghazi Al-Naymat, and Eshraq Al-Hawari. "Towards generating realistic SNMP-MIB dataset for network anomaly detection". In: *International Journal of Computer Science and Information Security* 14.9 (2016), p. 1162.

[3] Albert Gran Alcoz et al. "Aggregate-based congestion control for pulse-wave DDoS defense". In: *Proceedings of the ACM SIGCOMM 2022 Conference*. 2022, pp. 693–706.

[4] Fahd A Alhaidari and Alia Mohammed Alrehan. "A simulation work for generating a novel dataset to detect distributed denial of service attacks on Vehicular Ad hoc NETwork systems". In: *International Journal of Distributed Sensor Networks* 17.3 (2021), p. 15501477211000287.

[5] Sabah Alzahrani and Liang Hong. "Generation of DDoS attack dataset for effective IDS development and evaluation". In: *Journal of Information Security* 9.4 (2018), pp. 225–241.

[6] Manos Antonakakis et al. "Understanding the mirai botnet". In: *26th USENIX security symposium (USENIX Security 17)*. 2017, pp. 1093–1110.

[7] Apple Inc. *Activity Monitor User Guide*. `https://support.apple.com/en-gb/guide/activity-monitor/welcome/mac`, Last visit July 30, 2023. 2023.

[8] Sunny Behal and Krishan Kumar. "Characterization and Comparison of DDoS Attack Tools and Traffic Generators: A Review." In: *Int. J. Netw. Secur.* 19.3 (2017), pp. 383–393.

[9] Theophilus Benson and Balakrishnan Chandrasekaran. "Sounding the bell for improving Internet (of Things) security". In: *Proceedings of the 2017 Workshop on Internet of Things Security and Privacy*. 2017, pp. 77–82.

[10] Sajal Bhatia et al. "A framework for generating realistic traffic for Distributed Denial-of-Service attacks and Flash Events". In: *computers & security* 40 (2014), pp. 95–107.

[11] Nikola Blazic et al. "Implementation of SYN flood attack simulator in NS-3". In: *2018 Zooming Innovation in Consumer Technologies Conference (ZINC)*. IEEE. 2018, pp. 110–113.

[12] Richard R Brooks et al. "Distributed denial of service (DDoS): a history". In: *IEEE Annals of the History of Computing* 44.2 (2021), pp. 44–54.

[13] Calculator Academy Team. *Packets Per Second Calculator*. `https://calculator.academy/packets-per-second-calculator/`, Last visit July 27, 2023. 2023.

[14] Center for Applied Internet Data Analysis (CAIDA). *Dataset for "DDoS Attack 2007" - Request Form*. https://www.caida.org/catalog/datasets/request_user_info_forms/ddos_dataset_request/, Last visit March 18, 2023. 2010.

[15] Xu Chen et al. "Real-time DDoS Defense in 5G-Enabled IoT: A Multidomain Collaboration Perspective". In: *IEEE Internet of Things Journal* (2022).

[16] Ilya V Chugunkov et al. "Development of the algorithm for protection against DDoS-attacks of type pulse wave". In: *2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*. IEEE. 2018, pp. 292–294.

[17] Cisco Systems, Inc. *ISP 3-Tier Model*. https://www.thousandeyes.com/learning/techtorials/isp-tiers, Last visit August 2, 2023. 2023.

[18] Robertas Damasevicius et al. "LITNET-2020: An annotated real-world network flow dataset for network intrusion detection". In: *Electronics* 9.5 (2020), p. 800.

[19] DDoS-Guard. *Hidden threat of Pulse Wave DDoS attacks*. https://ddos-guard.net/en/blog/hidden-threat-of-pulse-wave-ddos-attacks, Last visit July 22, 2023. 2018.

[20] Ozgur Depren et al. "An intelligent intrusion detection system (IDS) for anomaly and misuse detection in computer networks". In: *Expert systems with Applications* 29.4 (2005), pp. 713–722.

[21] Rohan Doshi, Noah Apthorpe, and Nick Feamster. "Machine learning ddos detection for consumer internet of things devices". In: *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2018, pp. 29–35.

[22] Malin Eriksson and Victor Hallberg. "Comparison between JSON and YAML for data serialization". In: *The School of Computer Science and Engineering Royal Institute of Technology* (2011), pp. 1–25.

[23] Calvin Falter. *EDDD: Distributed DDoS Dataset Generator*. https://github.com/calvin-f/EDDD, Last visit July 22, 2023. 2023.

[24] Fortinet, Inc. *DoS Attack vs. DDoS Attack*. https://www.fortinet.com/resources/cyberglossary/dos-vs-ddos, Last visit February 23, 2023. 2023.

[25] Vladimir Galyaev et al. "Recent Trends in Development of DDoS Attacks and Protection Systems Against Them." In: *Int. J. Netw. Secur.* 21.4 (2019), pp. 635–647.

[26] ggouaillardet. *A system call failed during shared memory initialization*. https://github.com/open-mpi/ompi/issues/7393#issuecomment-882018321, Last visit August 4, 2023. 2021.

[27] GitHub user 'biojppm'. *rapidyaml*. https://github.com/biojppm/rapidyaml, Last visit July 15, 2023. 2023.

[28] GitHub user 'saulodamata'. *ns-3-http-traffic-generator*. https://github.com/saulodamata/ns-3-http-traffic-generator, Last visit July 15, 2023. 2020.

[29] GitHub users 'lhridder' and 'Wqrld'. *StopDDoS/packet-captures*. https://github.com/StopDDoS/packet-captures, Last visit August 1, 2023. 2023.

[30] Google. *Google Scholar*. https://scholar.google.com/, Last visit July 14, 2023. 2023.

[31] Arvin Hekmati, Eugenio Grippo, and Bhaskar Krishnamachari. "Large-scale urban iot activity data for ddos attack emulation". In: *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*. 2021, pp. 560–564.

[32] Tom Henderson. *Bug 1965 - restrictive assert in ECMP code*. https://www.nsnam.org/bugzilla/show_bug.cgi?id=1965, Last visit July 23, 2023. 2014.

[33] Tom Henderson. *Bug 667 - ECMP operation in global routing*. https://www.nsnam.org/bugzilla/show_bug.cgi?id=667#c16, Last visit July 23, 2023. 2017.

[34] KW Hove. "Automated DDoS Attack Fingerprinting by Mimicking the Actions of a Network Operator". B.S. thesis. University of Twente, 2019.

[35] IANA. *Internet Control Message Protocol (ICMP) Parameters*. https://www.iana.org/assignments/icmp-parameters/icmp-parameters.xhtml, Last visit July 22, 2023. 2023.

[36] Imperva. *Ping flood (ICMP flood)*. https://www.imperva.com/learn/ddos/ping-icmp-flood/, Last visit July 22, 2023. 2023.

[37] Imperva. *TCP SYN Flood*. https://www.imperva.com/learn/ddos/syn-flood/, Last visit July 22, 2023. 2023.

[38] Imperva. *UDP Flood*. https://www.imperva.com/learn/ddos/udp-flood/, Last visit July 22, 2023. 2023.

[39] Imperva. *Understanding Pulse Wave DDoS Attacks*. https://www.imperva.com/resources/resource-library/white-papers/understanding-pulse-wave-ddos-attacks/, Last visit July 22, 2023. 2017.

[40] Internet Engineering Task Force (IETF). *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks*. https://www.rfc-editor.org/rfc/rfc6550#section-6.2, Last visit March 7, 2023. 2012.

[41] Houssain Kettani and Robert M Cannistra. "On cyber threats to smart digital environments". In: *proceedings of the 2nd international conference on smart digital environment*. 2018, pp. 183–188.

[42] Houssain Kettani and Polly Wainwright. "On the top threats to cyber systems". In: *2019 IEEE 2nd international conference on information and computer technologies (ICICT)*. IEEE. 2019, pp. 175–179.

[43] Nickolaos Koroniotis et al. "Towards the development of realistic botnet dataset in the internet of things for network forensic analytics: Bot-iot dataset". In: *Future Generation Computer Systems* 100 (2019), pp. 779–796.

[44] Jeongeun Julie Lee and Maruti Gupta. "A new traffic model for current user web browsing behavior". In: *Intel corporation* (2007).

[45] Tasnuva Mahjabin et al. "A survey of distributed denial-of-service attack, prevention, and mitigation techniques". In: *International Journal of Distributed Sensor Networks* 13.12 (2017), p. 1550147717741463.

[46] Jelena Mirkovic and Peter Reiher. "A taxonomy of DDoS attack and DDoS defense mechanisms". In: *ACM SIGCOMM Computer Communication Review* 34.2 (2004), pp. 39–53.

[47] Robert Mitchell and Ing-Ray Chen. "A survey of intrusion detection techniques for cyber-physical systems". In: *ACM Computing Surveys (CSUR)* 46.4 (2014), pp. 1–29.

[48] Nour Moustafa and Jill Slay. "UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)". In: *2015 military communications and information systems conference (MilCIS)*. IEEE. 2015, pp. 1–6.

[49] ns-3 community. *Event-related memory consumption issue*. https://groups.google.com/g/ns-3-users/c/tEjTIJtDhYU/, Last visit July 27, 2023. 2023.

[50] ns-3 community. *Increase Memory Consumption - EnergyModel*. https://groups.google.com/g/ns-3-users/c/scoHXqFcjE0/m/lL4IhwfOCQAJ, Last visit July 27, 2023. 2018.

[51] ns-3 community. *ns-3-users*. https://groups.google.com/g/ns-3-users/, Last visit July 14, 2023. 2023.

[52] ns-3 community. *ns-3-users: Dynamic Target Address Change on OnOffApplication*. https://groups.google.com/g/ns-3-users/c/kfdMW6s9CjI/m/7wgqEM7nBgAJ, Last visit July 22, 2023. 2023.

[53] ns-3 community. *NS3 Support for Multiple paths*. https://groups.google.com/g/ns-3-users/c/njclO2klIr0/m/N6QgoORdkcUJ, Last visit July 23, 2023. 2014.

[54] nsnam. *Documentation*. https://www.nsnam.org/documentation/, Last visit July 14, 2023. 2023.

[55] nsnam. *ns-3: MPI for Distributed Simulation*. https://www.nsnam.org/docs/models/html/distributed.html, Last visit July 15, 2023. 2023.

[56] nsnam. *NS-3 Release 3.39*. https://gitlab.com/nsnam/ns-3-dev/-/tags/ns-3.39, Last visit July 14, 2023. 2023.

[57] nsnam. *ns3 Documentation*. https://www.nsnam.org/doxygen/, Last visit July 28, 2023. 2023.

[58] OpenAI. *ChatGPT*. https://chat.openai.com/, Last visit July 14, 2023. 2023.

[59] OpenSim Ltd. *OMNet++*. https://omnetpp.org/, Last visit July 14, 2023. 2023.

[60] Tommaso Pecorella. *Release memory when sockets are closed*. https://gitlab.com/nsnam/ns-3-dev/-/merge_requests/1515, Last visit July 22, 2023. 2023.

[61] PKiechl. *DPWS-PoC*. https://github.com/PKiechl/DPWS-PoC, Last visit August 4, 2023. 2023.

[62] Rastin Pries, Zsolt Magyari, and Phuoc Tran-Gia. "An HTTP web traffic model based on the top one million visited web pages". In: *Proceedings of the 8th Euro-NF Conference on Next Generation Internet NGI 2012*. IEEE. 2012, pp. 133–139.

[63] Tamara Radivilova, Lyudmyla Kirichenko, and Abed Saif Alghawli. "Entropy Analysis Method for Attacks Detection". In: *2019 IEEE International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC S&T)*. IEEE. 2019, pp. 443–446.

[64] Bruno Rodrigues, Thomas Bocek, and Burkhard Stiller. "Multi-domain DDoS mitigation based on blockchains". In: *Security of Networks and Services in an All-Connected World: 11th IFIP WG 6.6 International Conference on Autonomous Infrastructure, Management, and Security, AIMS 2017, Zurich, Switzerland, July 10-13, 2017, Proceedings 11*. Springer. 2017, pp. 185–190.

[65] Bruno Rodrigues et al. "Blockchain signaling system (BloSS): cooperative signaling of distributed denial-of-service attacks". In: *Journal of Network and Systems Management* 28 (2020), pp. 953–989.

[66] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. "Toward generating a new intrusion detection dataset and intrusion traffic characterization." In: *ICISSp* 1 (2018), pp. 108–116.

[67] Iman Sharafaldin et al. "Developing realistic distributed denial of service (DDoS) attack dataset and taxonomy". In: *2019 International Carnahan Conference on Security Technology (ICCST)*. IEEE. 2019, pp. 1–8.

[68] Swiss Internet Exchange. *Get connected*. https://www.swissix.ch/infrastructure/get-connected/, Last visit June 25, 2023. 2023.

[69] Swiss Internet Exchange. *SwissIX*. `https://www.swissix.ch/`, Last visit June 25, 2023. 2023.

[70] Swiss Internet Exchange. *What is an Internet Exchange Point (IXP)*. `https://www.swissix.ch/about/what-is-an-ixp/`, Last visit June 25, 2023. 2023.

[71] Aparna Tomar et al. "A Step Towards Generation of DoS/DDoS Attacks Dataset for Docker-Centric Computing". In: *International Journal of Mathematical, Engineering and Management Sciences* 7.1 (2022), p. 81.

[72] Imtiaz Ullah and Qusay H Mahmoud. "A scheme for generating a dataset for anomalous activity detection in iot networks". In: *Advances in Artificial Intelligence: 33rd Canadian Conference on Artificial Intelligence, Canadian AI 2020, Ottawa, ON, Canada, May 13–15, 2020, Proceedings 33*. Springer. 2020, pp. 508–520.

[73] Imtiaz Ullah and Qusay H Mahmoud. "A technique for generating a botnet dataset for anomalous activity detection in IoT networks". In: *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE. 2020, pp. 134–140.

[74] University of Zurich. *Department of Informatics - Communication Systems Group*. `https://www.csg.uzh.ch/csg/en/`, Last visit July 14, 2023. 2023.

[75] Matthias Wichtlhuber et al. "IXP scrubber: learning from blackholing traffic for ML-driven DDoS detection at scale". In: *Proceedings of the ACM SIGCOMM 2022 Conference*. 2022, pp. 707–722.

[76] Wireshark Foundation. *Wireshark*. `https://www.wireshark.org/`, Last visit August 3, 2023. 2023.

[77] Zeifman, Igal. *Attackers Use DDoS Pulses to Pin Down Multiple Targets*. `https://www.imperva.com/blog/pulse-wave-ddos-pins-down-multiple-targets/`, Last visit July 22, 2023. 2020.

Abbreviations

ACC	Aggregate-based Congestion Control
AS	Autonomous System
CLI	Command-line Interface
CSMA	Carrier Sense Multiple Access
CN	Central Network
CSG	Communication Systems Group
DoS	Denial of Service
DDoS	Distributed Denial of Service
ECMP	Equal Cost Multi-Path
HTTP	Hypertext Transfer Protocol
IDS	Intrusion Detection System
IoT	Internet of Things
IPS	Intrusion Prevention System
IX	Internet Exchange
IXP	Internet Exchange Point
LOIC	Low Orbit Ion Canon
ML	Machine Learning
MPI	Message Passing Interface
PCAP	Packet Capture
PPS	Packets Per Second
RAM	Random-Access Memory
TTL	Time-to-live
VANET	Vehicular ad Hoc Network
YAML	Yet Another Markup Language

List of Figures

List of Tables

Listings

Appendix A

Contents of the CD

The following list of deliverables is submitted for this master thesis:

- **Code**:
  - ZIP file containing the thesis' source code. Note that the source code is also made available on the project's GitHub repository [61].
- **Thesis**:
  - ZIP file containing the LaTeX source of the thesis
  - PDF of the thesis
  - Plain text file of the English abstract
  - Plain text file of the German abstract

Appendix B

Installation Guidelines

The installation guidelines are also present on the project's GitHub repository [61].

B.1 Distributed Pulse-Wave DDoS Simulator

The simulator requires that you have NS-3 installed.

1. Clone the NS-3 GitLab Repository:
   `git clone https://gitlab.com/nsnam/ns-3-dev.git`
2. Change into the repository directory:
   `cd ns-3-dev`
3. Select the release of your choice. This project was developed under version 3.38, thus your mileage may vary when using different versions:
   `git checkout -b ns-3.38-branch ns-3.38`

The simulator requires CMake and MPI to be installed on your system. On Mac OS you may use:

- `brew install cmake`
  to install CMake
- `brew install open-mpi`
  to install OpenMPI

The next step is to configure your NS-3 installation and ensure that it is functioning properly.

1. Configure your NS-3 installation. You may wish to use the `debug` build profile instead of the `optimized` profile if you plan to debug or develop. If you simply wish to run the simulator, then the optimized profile will yield significantly better performance.
   `./ns3 configure --enable-mpi --build-profile=optimized`

2. Once the configuration has finished, run the build script. This may take a moment to finish:
   `./ns3 build`

3. (Optional) Test that your installation is functional:
   `./test.py`

At this point you have a functioning NS-3 installation on your system and it is time to install the project's simulator files.

- Clone the project GitHub repository:
  `git clone https://github.com/PKiechl/DPWS-PoC.git`
- As a next step, a number of files have to be transferred to your NS-3 installation:
  1. Copy the contents of the `contrib` directory to the `contrib` directory of your NS-3 installation
  2. Copy the contents of the `scratch` directory to the `scratch` directory of your NS-3 installation
  3. Copy the contents of the `socket memory patch/src/internet/model` directory to the `src/internet/model` directory of your NS-3 installation and overwrite the files present there.
  4. Copy the contents of the `overwrites/src/internet/model` directory to the `src/internet/model` directory of your NS-3 installation and overwrite the file present there.
- As a last step, the build step must be executed again such that the newly transferred files take effect: `./ns3 build`

If you are on the Mac OS then you should consider altering your `TEMPDIR` as it has been known to experience truncation which can lead to errors when using MPI [26]. To circumvent this issue, run:
`export TMPDIR=/tmp`

In order to run the simulator you may use the `example_configuration.yaml` configuration or use your own configuration file. You can supply a number of optional commands when running the simulator:

- `--command-template="mpiexec -np {number_of_cores} %s"` allows you to make use of parallelization and control how many cores the simulator is allowed to use. Specify the number of cores as a number without the curly brackets.

- `--printConfiguration=true` provides a console printout of the entire configuration including optional values that you may not have explicitly specified in the configuration file.
- `--progressLogInterval={number}` allows you to control the interval in which the simulator will provide console output to indicate progress in the simulation timeline. The value is in seconds and represents simulation time and not execution time. Thus if you run larger scale use cases you may wish to use a lower value in order to get a better sense of progress. The default value is 15 seconds.
- `--printTopology=false` allows you to have the simulator not print the topology in the form of pairs of connected nodes at the end of the simulation run.

To execute a simulation run using all options use:

```
./ns3 run dpws --command-template="mpiexec -np 4 %s"
          -- --configFile="sample_config.yaml"
          --printConfiguration=true
          --progressLogInterval=5
          --printTopology=false
```

Due to the way the NS-3 run command operates, `command-template` must directly follow `dpws` with all other arguments lining up behind an additional `--` as shown above. The `configFile` parameter is required.

B.2 Evaluation Scripts

Clone the project repository:

1. `git clone https://github.com/PKiechl/DPWS-PoC.git`
2. `cd DPWS-PoC/evaluation`

Ensure you have Python 3 and pip3 installed on your system. These scripts were written with Python 3.11, thus your mileage may vary when using different versions.

1. Install Python 3.11 in the manner of your choice. On Mac OS you can use:
   `brew install python@3.11`
   which will also install pip3
2. If you wish to double check if you have the pip3 packet manager installed:
   `python3 -m ensurepip`

Set up a virtual environment for the installation of the Python requirements.

1. Ensure that you have virtualenv installed. To install it use:
   `pip3 install virtualenv`
   You may also use alternatives such as conda, though this guide will assume that you use virtualenv.

2. Create a new virtual environment. In this guide it is called `venv` though you may elect to use a different name:
   `python3 -m virtualenv venv`

3. Activate the new virtual environment:
   `source venv/bin/activate`

4. Install the requirements:
   `pip3 install -r requirements.txt`

In order to run the script of your choice you may use:
`python3 path/to/script.py`

For example, to gather port statistics from a PCAP file run:
`python3 evaluation/traffic/calculate_port_number_percentages.py`

Be cognizant of the fact that the evaluation scripts rely on files being present and other parameters being defined within the script. You will have to adjust those in the scripts before executing them.

Appendix C

Additional Figures

Figure C.1: Autonomous System Class Diagram, Larger Scale

Appendix D

Evaluation Scripts

The script shown in Listing D.1 is used to create stack plots of the specified PCAP file's attack traffic, such that the contributions to the attack of the individual attacker nodes can be highlighted.

Listing D.1: plot_attacker_and_protocol_keyed_traffic_at_ixp_node.py

```
1  import matplotlib.pyplot as plt
2  from matplotlib.patches import Patch
3  import numpy as np
4  from scapy.all import rdpcap
5  from collections import defaultdict
6  import warnings
7
8  def process_pcap(pcap, dr_tcp_global, pv_tcp_global, dr_udp_global, pv_udp_global,
       dr_icmp_global, pv_icmp_global,
9                   attackers, max_time):
10     for pkt in pcap:
11         if 'IP' in pkt:
12             ip = pkt['IP']
13
14             # don't process packets whose ip.src does not match any of the attackers
15             if ip.src not in attackers:
16                 continue
17
18             time_bin_index = int(pkt.time)
19             # only store information for range covered by provided max_time (end_time)
20             if time_bin_index <= max_time:
21                 # update data rate and packet count for src--time_bin pair for given
       protocol
22                 if ip.proto == 1:
23                     # ICMP
24                     dr_icmp_global[ip.src][time_bin_index] += len(pkt)
25                     pv_icmp_global[ip.src][time_bin_index] += 1
26                 elif ip.proto == 6:
27                     # TCP
28                     dr_tcp_global[ip.src][time_bin_index] += len(pkt)
29                     pv_tcp_global[ip.src][time_bin_index] += 1
30                 elif ip.proto == 17:
31                     # UDP
```

```
32                      dr_udp_global[ip.src][time_bin_index] += len(pkt)
33                      pv_udp_global[ip.src][time_bin_index] += 1
34                  else:
35                      warnings.warn("Found packet that does not fit protocols (UDP, TCP,
        ICMP)", pkt)
36              else:
37                  warnings.warn("Found packet without IP layer:", pkt)
38
39  def rearrange_into_array(attackers, dr_dict, dr_arr, pv_dict, pv_arr):
40      for atk in attackers:
41          # check if atk actually was present in any packet (both dicts share same keys)
42          if atk in dr_dict.keys():
43              dr_arr.append([dr / 1e6 * 8 for dr in dr_dict[atk]])
44              pv_arr.append(pv_dict[atk])
45
46  def plot_traffic(pcap_file_list, attacker_ip_list, end_time, plot_file_name):
47      # establish x-axis dimension (time elapsed)
48      time_array = np.arange(end_time + 1)  # 1 second time "bins". range is exclusive
        at the end, thus + 1
49
50      # collections for data rates and packet volume per protocol
51      dr_tcp_per_attacker_per_time_bin = defaultdict(lambda: [0] * len(time_array))
52      pv_tcp_per_attacker_per_time_bin = defaultdict(lambda: [0] * len(time_array))
53      dr_udp_per_attacker_per_time_bin = defaultdict(lambda: [0] * len(time_array))
54      pv_udp_per_attacker_per_time_bin = defaultdict(lambda: [0] * len(time_array))
55      dr_icmp_per_attacker_per_time_bin = defaultdict(lambda: [0] * len(time_array))
56      pv_icmp_per_attacker_per_time_bin = defaultdict(lambda: [0] * len(time_array))
57
58      # perform accumulation across all files
59      for file in pcap_file_list:
60          print("Processing file:", file)
61          pcap = rdpcap(file)
62          process_pcap(pcap,
63                       dr_tcp_per_attacker_per_time_bin,
64                       pv_tcp_per_attacker_per_time_bin,
65                       dr_udp_per_attacker_per_time_bin,
66                       pv_udp_per_attacker_per_time_bin,
67                       dr_icmp_per_attacker_per_time_bin,
68                       pv_icmp_per_attacker_per_time_bin,
69                       attacker_ip_list,
70                       end_time)
71
72      # convert dictionaries to nested arrays for plotting, convert data rate form Bytes
        /s to MBit/s -> div by 1e6 * 8
73      # also perform reordering such that order in nested array follows order of
        attackers in initial attacker_ip_list
74      dr_tcp_nested_ordered = []
75      pv_tcp_nested_ordered = []
76      dr_udp_nested_ordered = []
77      pv_udp_nested_ordered = []
78      dr_icmp_nested_ordered = []
79      pv_icmp_nested_ordered = []
80
81      rearrange_into_array(attacker_ip_list, dr_tcp_per_attacker_per_time_bin,
        dr_tcp_nested_ordered,
82                           pv_tcp_per_attacker_per_time_bin, pv_tcp_nested_ordered)
```

```
    rearrange_into_array(attacker_ip_list, dr_udp_per_attacker_per_time_bin, dr_udp_nested_ordered,
                         pv_udp_per_attacker_per_time_bin, pv_udp_nested_ordered)
    rearrange_into_array(attacker_ip_list, dr_icmp_per_attacker_per_time_bin, dr_icmp_nested_ordered,
                         pv_icmp_per_attacker_per_time_bin, pv_icmp_nested_ordered)

    # perform per-time-bin sum across all attacker nodes for the supplementary line-plots
    dr_summed = np.zeros(len(time_array))
    pv_summed = np.zeros(len(time_array))
    for time in time_array:
        for lst in dr_tcp_nested_ordered:
            dr_summed[time] += lst[time]
        for lst in dr_udp_nested_ordered:
            dr_summed[time] += lst[time]
        for lst in dr_icmp_nested_ordered:
            dr_summed[time] += lst[time]

        for lst in pv_tcp_nested_ordered:
            pv_summed[time] += lst[time]
        for lst in pv_udp_nested_ordered:
            pv_summed[time] += lst[time]
        for lst in pv_icmp_nested_ordered:
            pv_summed[time] += lst[time]

    colors_stack_TCP = ['#162258', '#40437e', '#6768a7', '#908fd2', '#d7d4ff']
    colors_stack_UDP = ['#003804', '#21652f', '#53955b', '#84c88b', '#b8febd']
    colors_stack_ICMP = ['#3e2707', '#6b4f2f', '#9a7a59', '#cba986', '#ffdab5']

    # https://stackoverflow.com/a/63741687
    p_TCP_1 = Patch(facecolor=colors_stack_TCP[0], edgecolor='black')
    p_TCP_2 = Patch(facecolor=colors_stack_TCP[1], edgecolor='black')
    p_TCP_3 = Patch(facecolor=colors_stack_TCP[2], edgecolor='black')
    p_TCP_4 = Patch(facecolor=colors_stack_TCP[3], edgecolor='black')
    p_TCP_5 = Patch(facecolor=colors_stack_TCP[4], edgecolor='black')

    p_UDP_1 = Patch(facecolor=colors_stack_UDP[0], edgecolor='black')
    p_UDP_2 = Patch(facecolor=colors_stack_UDP[1], edgecolor='black')
    p_UDP_3 = Patch(facecolor=colors_stack_UDP[2], edgecolor='black')
    p_UDP_4 = Patch(facecolor=colors_stack_UDP[3], edgecolor='black')
    p_UDP_5 = Patch(facecolor=colors_stack_UDP[4], edgecolor='black')

    p_ICMP_1 = Patch(facecolor=colors_stack_ICMP[0], edgecolor='black')
    p_ICMP_2 = Patch(facecolor=colors_stack_ICMP[1], edgecolor='black')
    p_ICMP_3 = Patch(facecolor=colors_stack_ICMP[2], edgecolor='black')
    p_ICMP_4 = Patch(facecolor=colors_stack_ICMP[3], edgecolor='black')
    p_ICMP_5 = Patch(facecolor=colors_stack_ICMP[4], edgecolor='black')


    # plot 1: stackplot of dr per attacker node with pv_summed line plot
    fig, ax = plt.subplots()
    # stackplots for data rate
    ax.stackplot(time_array, dr_tcp_nested_ordered, colors=colors_stack_TCP)
    ax.stackplot(time_array, dr_udp_nested_ordered, colors=colors_stack_UDP)
    ax.stackplot(time_array, dr_icmp_nested_ordered, colors=colors_stack_ICMP)
```

```
136     ax.legend(bbox_to_anchor=(1.15, 0.85), loc="upper left",
137               handles=[p_TCP_5, p_TCP_4, p_TCP_3, p_TCP_2, p_TCP_1,
138                        p_UDP_5, p_UDP_4, p_UDP_3, p_UDP_2, p_UDP_1,
139                        p_ICMP_5, p_ICMP_4, p_ICMP_3, p_ICMP_2, p_ICMP_1],
140               labels=["", "", "", "", "",
141                       "", "", "", "", "",
142                       "Attacker 5 (TCP / UDP / ICMP)",
143                       "Attacker 4 (TCP / UDP / ICMP)",
144                       "Attacker 3 (TCP / UDP / ICMP)",
145                       "Attacker 2 (TCP / UDP / ICMP)",
146                       "Attacker 1 (TCP / UDP / ICMP)"],
147               ncol=3, handletextpad=0.5, handlelength=2.5, columnspacing=-0.5)
148
149     ax.set_xlabel('Time [s]')
150     ax.set_ylabel('Data Rate [MBit/s]')
151     ax.set_title('Data Rate per Attacker Node over Time')
152     # line plot for packet volume
153     ax_alt = ax.twinx()
154     ax_alt.plot(time_array, pv_summed, label="Combined Packet Volume", linewidth=1.25,
          color="red")
155     ax_alt.legend(bbox_to_anchor=(1.15, 1), loc="upper left")
156     ax_alt.set_ylim(ymin=0)  # prevent entire subplot from "floating" slightly above
        the bottom of the graph
157     ax_alt.set_ylabel("Packet Volume [Pkt/s]")
158     # save and show
159     plt.savefig(f"{plot_file_name}__data_rate.pdf", format="pdf", bbox_inches="tight")
160     plt.show()
161
162
163     # # plot 2: stackplot of pv per attacker node with dr_summed line plot
164     fig2, ax2 = plt.subplots()
165     # stackplots for packet volume
166     ax2.stackplot(time_array, pv_tcp_nested_ordered, colors=colors_stack_TCP)
167     ax2.stackplot(time_array, pv_udp_nested_ordered, colors=colors_stack_UDP)
168     ax2.stackplot(time_array, pv_icmp_nested_ordered, colors=colors_stack_ICMP)
169     ax2.legend(bbox_to_anchor=(1.15, 0.85), loc="upper left",
170                handles=[p_TCP_5, p_TCP_4, p_TCP_3, p_TCP_2, p_TCP_1,
171                         p_UDP_5, p_UDP_4, p_UDP_3, p_UDP_2, p_UDP_1,
172                         p_ICMP_5, p_ICMP_4, p_ICMP_3, p_ICMP_2, p_ICMP_1],
173                labels=["", "", "", "", "",
174                        "", "", "", "", "",
175                        "Attacker 5 (TCP / UDP / ICMP)",
176                        "Attacker 4 (TCP / UDP / ICMP)",
177                        "Attacker 3 (TCP / UDP / ICMP)",
178                        "Attacker 2 (TCP / UDP / ICMP)",
179                        "Attacker 1 (TCP / UDP / ICMP)"],
180                ncol=3, handletextpad=0.5, handlelength=2.5, columnspacing=-0.5)
181     ax2.set_xlabel('Time [s]')
182     ax2.set_ylabel('Packet Volume [Pkt/s]')
183     ax2.set_title('Packet Volume per Attacker Node over Time')
184     # line plot for data rate
185     ax2_alt = ax2.twinx()
186     ax2_alt.plot(time_array, dr_summed, label="Combined Data Rate", linewidth=1.25,
          color="red")
187     ax2_alt.legend(bbox_to_anchor=(1.15, 1), loc="upper left")
```

```
188     ax2_alt.set_ylim(ymin=0)  # prevent entire subplot from "floating" slightly above
          the bottom of the graph
189     ax2_alt.set_ylabel("Data Rate [MBit/s]")
190     # save and show
191     plt.savefig(f"{plot_file_name}__packet_volume.pdf", format="pdf", bbox_inches="
          tight")
192     plt.show()
193
194
195 # List pcap file paths
196 pcap_files = ["filename.pcap"]
197 # List ip addresses of the attacker nodes
198 attacker_node_ips = ["192.168.1.2", "192.168.2.2", "192.168.3.2", "192.168.4.2", "
          192.168.5.2"]
199 plot_traffic(pcap_files, attacker_node_ips, 220, "output_filename")
```

The code shown in Listing D.2 is used to calculate a given attack traffic pulse's port utilization metrics.

Listing D.2: calculate_port_number_percentages.py

```
1  from scapy.all import *
2  import warnings
3
4
5  def sum_per_port_percentage(packets, target_ip, source_ip_list, end_time, start_time):
6      # use target_ip and source_ip_list to narrow down to just attack traffic in one
         pulse
7
8      source_ports = {}
9      dest_ports = {}
10     total_packets = 0
11
12     for pkt in packets:
13         if 'IP' in pkt:
14             # filter packets to only specified pulse and duration
15             if pkt['IP'].src not in source_ip_list:
16                 continue
17             if pkt['IP'].dst != target_ip:
18                 continue
19             if pkt.time > end_time or pkt.time < start_time:
20                 continue
21
22             # grab transport layer packet port counts
23             src_port = -1
24             dst_port = -1
25             if 'TCP' in pkt:
26                 src_port = pkt['TCP'].sport
27                 dst_port = pkt['TCP'].dport
28             elif 'UDP' in pkt:
29                 src_port = pkt['UDP'].sport
30                 dst_port = pkt['UDP'].dport
31
32             if src_port == -1 or dst_port == -1:
33                 warnings.warn("found packet without port, double check your time
       stamps")
34             else:
```

```
35                 total_packets += 1
36                 source_ports[src_port] = source_ports.get(src_port, 0) + 1
37                 dest_ports[dst_port] = dest_ports.get(dst_port, 0) + 1
38     return source_ports, dest_ports, total_packets
39
40
41 def calculate_port_percentages(filename, target_ip, source_ip_list, start_time,
       end_time, random_trim_threshold):
42     packets = rdpcap(filename)
43     source_ports, dest_ports, total_packets = sum_per_port_percentage(packets,
       target_ip, source_ip_list, end_time,
44                                                                       start_time)
45
46     source_port_percentages = {port: (count / total_packets) * 100 for port, count in
       source_ports.items()}
47     dest_port_percentages = {port: (count / total_packets) * 100 for port, count in
       dest_ports.items()}
48
49     # sorting
50     sorted_src_count = dict(sorted(source_ports.items(), key=lambda item: item[1],
       reverse=True))
51     sorted_src_percent = dict(sorted(source_port_percentages.items(), key=lambda item:
        item[1], reverse=True))
52     sorted_dest_count = dict(sorted(dest_ports.items(), key=lambda item: item[1],
       reverse=True))
53     sorted_dest_percent = dict(sorted(dest_port_percentages.items(), key=lambda item:
       item[1], reverse=True))
54
55     # remove those with percentages below threshold
56     filtered_sorted_src_count = {key: value for key, value in sorted_src_count.items()
        if value >= random_trim_threshold*total_packets}
57     filtered_sorted_dest_count = {key: value for key, value in sorted_dest_count.items
       () if value >= random_trim_threshold*total_packets}
58     filtered_sorted_src_percent = {key: value for key, value in sorted_src_percent.
       items() if value >= random_trim_threshold}
59     filtered_sorted_dest_percent = {key: value for key, value in sorted_dest_percent.
       items() if value >= random_trim_threshold}
60
61     # count ports with percentages below threshold, use as indication for
       randomization
62     src_ports_below_threshold = len(sorted_src_percent) - len(
       filtered_sorted_src_percent)
63     dest_ports_below_threshold = len(sorted_dest_percent) - len(
       filtered_sorted_dest_percent)
64
65     # treat low-count/low-percent as randomized -> set trim threshold to change the
       cut off
66     # to determiziation randomiziation percent, the percentages ABOVE the threshold are
       subtracted from 1, yielding the remaining
67     src_port_random_percent = round(100 - sum(filtered_sorted_src_percent.values()),
       2)
68     dest_port_random_percent = round(100 - sum(filtered_sorted_dest_percent.values()),
        2)
69
70     print(f"total packets considered: {total_packets}\n")
71
```

```
72      print(f"source ports counts: {filtered_sorted_src_count}")
73      print(f"source ports percentage breakdown: {filtered_sorted_src_percent}")
74      print(f"number of individual source port numbers below occurrence threshold: {
          src_ports_below_threshold}")
75      print(f"percent of source ports that are randomized: {src_port_random_percent}\n")
76
77      print(f"destination ports counts: {filtered_sorted_dest_count}")
78      print(f"destination ports percentage breakdown: {filtered_sorted_dest_percent}")
79      print(f"number of individual destination port numbers below occurrence threshold:
          {dest_ports_below_threshold}")
80      print(f"percent of destination ports that are randomized: {
          dest_port_random_percent}\n")
81
82
83
84  src_list = ["192.168.1.2", "192.168.2.2", "192.168.3.2", "192.168.4.2", "192.168.5.2"]
85  # use the start_time and end_time to restrict the analysis to a sepcific part of the
        packet. In combination with the
86  # ip address arguments you can narrow down the analysis to a specific pulse
87  # use trim threshold to determine which percentages are considered low enough to be
        random
88  calculate_port_percentages("VAR1__IXP2-to-AS2____-1-2.pcap", "192.173.1.2", src_list,
        60, 90, 0.01)
```

The script shown in Listing D.3 serves the purpose of extracting statistics about packet sizes, packet volume and data rates from PCAP traces.

Listing D.3: calculate_per_packet_stats.py

```
1   from scapy.all import *
2
3   def calculate_raw_stats(packets, start, end, target, sources):
4       data_per_ip = {}  # data per ip address
5       data_sizes = {} # counts per packet-size
6
7       for pkt in packets:
8           if "IP" in pkt:
9               # filter packets to only specified pulse and duration
10              if pkt['IP'].src not in sources:
11                  continue
12              if pkt['IP'].dst != target:
13                  continue
14              if pkt.time > end or pkt.time < start:
15                  continue
16
17              src_ip = pkt["IP"].src
18              size = len(pkt)
19
20              # accumulate per IP
21              if src_ip in data_per_ip:
22                  data_per_ip[src_ip]['total_size'] += size
23                  data_per_ip[src_ip]['total_packets'] += 1
24              else:
25                  data_per_ip[src_ip] = {'total_size': size, 'total_packets': 1}
26              # count per size
27              if size in data_sizes:
28                  data_sizes[size] += 1
```

```
            else:
                data_sizes[size] = 1

    return data_per_ip, data_sizes

def calculate_values(filename, start, end, target, sources):
    packets = rdpcap(filename)
    pv_dr_per_ip, pkt_sizes = calculate_raw_stats(packets, start, end, target, sources
      )

    for ip, data in pv_dr_per_ip.items():
        data_bytes = data['total_size']
        num_pkts = data['total_packets']
        avg_mps = ((data_bytes / (end-start)) / 1000000) * 8  # convert to Mbps
        avg_pps = num_pkts / (end-start)

        print(f"\nIP: {ip}")
        print(f"\t\tAverage DR: {avg_mps:.4f} Mbps")
        print(f"\t\tAverage PV: {avg_pps:.4f} Pps")

    pkt_count = sum(pkt_sizes.values())
    print("\nPacket Size Percentages:")
    for size, data in pkt_sizes.items():
        print(f"{size}: {data/pkt_count*100:.2f} %")

    # sum packet counts and accumulated bytes
    total_pv = 0
    total_bytes = 0
    for ip, data in pv_dr_per_ip.items():
        total_pv += data['total_packets']
        total_bytes += data['total_size']

    print("\nAverages:")
    print(f"\t\tOverall Average DR: {((total_bytes / (end-start)) / 1000000) * 8:.4f}
      Mbps")
    print(f"\t\tOverall Average PV: {total_pv / (end-start):.4f}")

sources=["192.168.1.2", "192.168.2.2", "192.168.3.2", "192.168.4.2", "192.168.5.2"]
calculate_values("filename.pcap", 180, 210, "192.173.1.2", sources)
```

Listing D.4 shows the code used for plotting the overall traffic pattern, filtered by IP addresses as data rate over time.

Listing D.4: plot_traffic_at_ixp_node.py

```
from scapy.all import *
import matplotlib.pyplot as plt
import numpy as np
import warnings


def process_pcap(pcap, packet_data_rate_global, ip_src_filter):
    for pkt in pcap:
        if 'IP' in pkt:
            ip = pkt['IP']
            # only process packets that are in filter list
            if ip.src not in ip_src_filter:
```

```
13                 continue
14
15             time_bin_index = int(pkt.time)
16             packet_data_rate_global[time_bin_index] += len(pkt)  # len(packet) gives
       packet size in bytes
17         else:
18             warnings.warn("Found packet without IP layer:", pkt)
19
20 def plot_traffic_data_rate(pcap_file_list, end_time, plot_file_name,
       filter_ip_src_list=None, force_y_lim=[]):
21     # mutable default argument should not be a problem here, but PyCharm is not a fan,
        so here we go :)
22     if filter_ip_src_list is None:
23         filter_ip_src_list = []
24
25     time_array = np.arange(end_time + 1)  # 1 second time "bins". range is exclusive
       at the end, thus + 1
26     packet_data_rate = np.zeros(len(time_array))  # bytes, stored per seconds, thus
       effectively yielding data rate
27
28     for file in pcap_file_list:
29         print("Processing file:", file)
30         pcap = rdpcap(file)
31         process_pcap(pcap, packet_data_rate, filter_ip_src_list)
32
33     # convert bytes go Mbit/s -> div by 1e6 then times 8
34     packet_data_rate = [dr / 1e6 * 8 for dr in packet_data_rate]
35
36     fig, ax = plt.subplots()
37     ax.plot(time_array, packet_data_rate, label="Attack Traffic", linewidth=0.75,
       alpha=0.9)
38     ax.legend(loc='lower right')
39     ax.set_xlabel('Time [s]')
40     ax.set_ylabel('Data Rate [Mbit/s]')
41     if len(force_y_lim) == 2:
42         ax.set_ylim(force_y_lim)
43     ax.set_title('Attack Traffic Data Rate over Time')
44     plt.savefig(plot_file_name, format="pdf", bbox_inches="tight")
45     plt.show()
46
47 pcap_files = ["file_name.pcap"]
48 filter_src_ip = ["192.168.1.2", "192.168.2.2", "192.168.3.2", "192.168.4.2", "
       192.168.5.2"]
49 plot_traffic_data_rate(pcap_files, 600, "VAR1_newCols.pdf", filter_ip_src_list=
       filter_src_ip)
```

The script shown in Listing D.5 is used to capture RAM and CPU usage of the host machine.

Listing D.5: capture-system-performance.py

```
1 import psutil
2 import csv
3 import time
4 import signal
5 import sys
6
```

```
7
8  def measure_system_resource_use():
9      # RAM % and CPU %
10     measurements = []
11
12     def signal_handler(sig, frame):
13         # https://stackoverflow.com/a/1112350
14         # capture program interrupt and write measurements to a CSV file before
       terminating
15         write_to_csv(measurements)
16         sys.exit(0)
17     # register signal handler
18     signal.signal(signal.SIGINT, signal_handler)
19
20     while True:
21         # Get current system time to synch up with execution of ns-3 simulation run
22         time_stamp = time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())
23
24         # Get CPU usage
25         cpu_percent = psutil.cpu_percent(interval=1)
26
27         # Get RAM usage
28         ram = psutil.virtual_memory()
29         ram_percent = ram.percent
30
31         # Create a dictionary for the measurements including the timestamp
32         measurement = {
33             "time_stamp": time_stamp,
34             "cpu_percent": cpu_percent,
35             "ram_percent": ram_percent,
36         }
37
38         # Append the measurement to the list
39         measurements.append(measurement)
40
41         # Print the measurements
42         print(f"Time: ${time_stamp}")
43         print(f"CPU Usage: {cpu_percent}%")
44         print(f"RAM Usage: {ram_percent}% ")
45
46         # Wait for 1 second before measuring again
47         # Note: this is very likely to have some slight timing drift. Additionally, it
       actually does not measure
48         # every second, but rather performs the measurements, then waits 1 second.
       this seems to result in more
49         # of an every 2 seconds measurements. This is perfectly fine, given that the
       goal is not to have measurements
50         # happening in perfect synch, but rather a steady stream of measurements such
       that trends over time can be
51         # observed.
52         time.sleep(1)
53
54
55 def write_to_csv(measurements):
56     # define fieldnames CSV
57     fieldnames = ["time_stamp", "cpu_percent", "ram_percent"]
```

```
58
59      # prompt for filename
60      filename = input("Enter the output filename (without file extension): ")
61      filename = filename+".csv"
62
63      # write to csv
64      with open(filename, mode="w", newline="") as file:
65          writer = csv.DictWriter(file, fieldnames=fieldnames)
66          # header row written separately
67          writer.writeheader()
68          writer.writerows(measurements)
69
70      print(f"Performance Data written to: '{filename}'")
71
72
73  measure_system_resource_use()
```

Listing D.6 contains the script used to plot multiple memory-consumption curves in one figure.

Listing D.6: multiplot-system-performance.py

```
1  # neglecting CPU in this plot because it is basically always maxed anyway
2
3  import csv
4  import matplotlib.pyplot as plt
5  from datetime import datetime
6
7  def create_plot_data_for_file(filename, idle_ram_avg, time_frame_start,
       time_frame_stop):
8      # perform full file read
9      timestamps = []
10     ram_percentages = []
11
12     with open(filename, mode="r") as file:
13         reader = csv.DictReader(file)
14         for row in reader:
15             timestamps.append(datetime.strptime(row["time_stamp"], "%Y-%m-%d %H:%M:%S"
       ))
16             ram_percentages.append(float(row["ram_percent"]))
17
18     # narrow down to specified time-frame & express timestamps as time elapsed in
       seconds
19     start_time = datetime.strptime(time_frame_start, "%Y-%m-%d %H:%M:%S")
20     end_time = datetime.strptime(time_frame_stop, "%Y-%m-%d %H:%M:%S")
21
22     to_plot__time_elapsed = []
23     start_index = -1
24     end_index = -1
25
26     for i, stamp in enumerate(timestamps):
27         # just to be safe we operate with the day as well. I sincerely hope none of
       the tests will for that long
28         # though...
29         if start_time <= stamp <= end_time:
30             diff = int((stamp - start_time).total_seconds())
```

```
            # int() to trim the decimal positions. datetime as used in this project does not consider sub-second
            # time, so the diff is always essentially just an integer with a .0
            to_plot__time_elapsed.append(diff)

            if start_index == -1:
                # take note of the first index to be considered in the cpu/ram data lists
                start_index = i
            end_index = i

    # extract relevant sub-lists of the cpu/ram data, based on the relevant time-frame (upper slice index is exclusive,
    # thus +1)
    to_plot__ram_percentages = ram_percentages[start_index:end_index + 1]

    # subtract the idle system averages to get isolated costs of just the simulation
    to_plot__ram_percentages = [el - idle_ram_avg for el in to_plot__ram_percentages]

    print(f"max ram for file {filename}: {max(to_plot__ram_percentages)}")

    return to_plot__time_elapsed, to_plot__ram_percentages

def plot_system_load(filename_list, timestamp_list, idle_ram_avg_list, plot_labels, plot_name, plot_title, force_full_y_axis=False, overwrite_legend_position=""):

    plt.title(plot_title)
    plt.xlabel("Time Elapsed [s]")
    plt.ylabel("RAM Usage [%]")

    for index, file in enumerate(filename_list):
        print(f"processing file: {file}")
        x, y = create_plot_data_for_file(file, idle_ram_avg_list[index], timestamp_list[index][0], timestamp_list[index][1])
        plt.plot(x, y, label=plot_labels[index])

    if overwrite_legend_position != "":
        plt.legend(loc=overwrite_legend_position)
    else:
        plt.legend()
    plt.locator_params(axis='x', nbins=15)  # Setting the number of ticks
    plt.xticks(rotation=45)
    if force_full_y_axis:
        plt.yticks([0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100])
    plt.tight_layout()

    # Save and show
    plt.savefig(f"{plot_name}.pdf", format="pdf", bbox_inches="tight")
    plt.show()

files = ["../../SC2_2p4Mib.csv", "../../SC2_2p4Mib_with_socket.csv"]
labels = ["SC2_HPV","SC2_HPV_PS"]
timestamps = [("2023-07-28 14:25:30", "2023-07-28 14:48:01"),("2023-07-28 14:50:58", "2023-07-28 15:07:21")]
ram_idle_avgs = [15.1,14.6]
plot_system_load(files, timestamps, ram_idle_avgs,  labels, "SC2p4_w_and_wo_sink", "
```

```
      Impact of UDP Packet Sink on Memory Consumption")
```

The code shown in Listing D.7 was used to demonstrate the reliability issues faced with the CSMA channel described in Section 5.4.0.1.

Listing D.7: Non-Prototype Related CSMA Showcase

```
#include "ns3/applications-module.h"
#include "ns3/core-module.h"
#include "ns3/internet-module.h"
#include "ns3/network-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/csma-module.h"


using namespace ns3;

NS_LOG_COMPONENT_DEFINE("OnOffExample");

int
main(int argc, char* argv[])
{
    CommandLine cmd;
    cmd.Parse(argc, argv);

    Time::SetResolution(Time::NS);
    LogComponentEnable("OnOffExample", LOG_LEVEL_INFO);

    // Create nodes
    NodeContainer nodes;
    nodes.Create(2);

    // Create links

    // use for point-to-point
    //    PointToPointHelper pointToPoint;
    //    pointToPoint.SetDeviceAttribute("DataRate", StringValue("500Gbps"));
    //    pointToPoint.SetChannelAttribute("Delay", StringValue("2ms"));

    // use for CSMA
    CsmaHelper csma;
    csma.SetChannelAttribute("DataRate", StringValue("500Gbps"));
    csma.SetChannelAttribute("Delay", StringValue("2ms"));

    NetDeviceContainer devices;
    // use for CSMA
    devices = csma.Install(nodes);
    // use for point-to-point
    //    devices = pointToPoint.Install(nodes);

    // Install internet stack
    InternetStackHelper stack;
    stack.Install(nodes);

    // Assign IP addresses
    Ipv4AddressHelper address;
    address.SetBase("10.1.1.0", "255.255.255.0");
```

```
51
52      Ipv4InterfaceContainer interfaces = address.Assign(devices);
53
54      // Create OnOff application
55      uint16_t port = 9;
56      OnOffHelper onOffHelper("ns3::UdpSocketFactory",
57                              Address(InetSocketAddress(interfaces.GetAddress(1), port))
          ↪ );
58      onOffHelper.SetAttribute("OnTime", StringValue("ns3::ConstantRandomVariable[
          ↪ Constant=5]"));
59      onOffHelper.SetAttribute("OffTime", StringValue("ns3::ConstantRandomVariable[
          ↪ Constant=10]"));
60      onOffHelper.SetAttribute("DataRate", StringValue("50Mbps"));
61      onOffHelper.SetAttribute("PacketSize", UintegerValue(1024));
62
63      ApplicationContainer apps = onOffHelper.Install(nodes.Get(0));
64      apps.Start(Seconds(0.0));
65      apps.Stop(Seconds(20.0));
66
67      // Create packet sink application
68      PacketSinkHelper packetSinkHelper("ns3::UdpSocketFactory",
69                                        Address(InetSocketAddress(Ipv4Address::GetAny(),
          ↪  port)));
70      apps = packetSinkHelper.Install(nodes.Get(1));
71      apps.Start(Seconds(0.0));
72      apps.Stop(Seconds(20.0));
73
74      // use for CSMA
75      csma.EnablePcap("on_off_test_csma", devices.Get(1), false);
76      // use for point-to-point
77      //    pointToPoint.EnablePcap("on_off_test", devices.Get(1), false);
78
79      Simulator::Run();
80      Simulator::Stop(Seconds(20.0));
81      Simulator::Destroy();
82      return 0;
83  }
```