Denial of Wallet: Analysis of a Looming Threat and Novel Solution for Mitigation using Image Classification

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Daniel Kelly

Supervisors: Dr. Enda Barrett
Dr. Frank Glavin

School of Computer Science
College of Science and Engineering
University of Galway

This thesis is submitted for the degree of
Doctor of Philosophy

University of Galway August 2023

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this thesis are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This thesis is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Daniel Kelly
August 2023

Acknowledgements

Firstly, I would like to thank my supervisors Dr. Enda Barrett and Dr. Frank Glavin who offered me their unwavering guidance over the Ph.D. process. It was often a struggle to maintain motivation, especially given global events, but they were always reassuring and offered their knowledge and opinions when I needed them.

I thank Dr. Barrett for convincing me to undertake a Ph.D. in the first place, something I would have never considered possible otherwise.

I thank Dr. Glavin for his meticulous eye for detail, without which the quality of my papers would be embarrassing to submit.

I thank my previous colleagues at Computer DISC. I am happy to have started my Ph.D. working at DISC, I believe we made a difference to many of the students who came to us. I am grateful for their continued hospitality and friendship.

I would like to thank my partner Órla, who put up with my near constant grumbles and frustrations. I wish her all the best for her own Ph.D.

Finally, I thank my family for having confidence in me to achieve this goal. Retreats to the family home were the perfect way to refresh the mind.

This project was funded by the Irish Research Council for the last two years 2021-2023 under project ID GOIPG/2021/288

Abstract

Serverless Computing is a powerful cloud-based architecture for the creation of applications. It boasts incredible scalability by running processes on a vast network of edge nodes. It decreases the time to deployment, as the developer no longer needs to programme a traditional server-side back-end, only having to focus on the application's business logic. Serverless applications are billed by counting the number of invocations a function receives in conjunction with its memory allocation. This means that there is no need to pay for the provision of a server that constantly runs in the background when it may only receive a small number of requests per month. These unique selling points, when used in the intended way, can drastically reduce operational costs. However, it has given rise to a potential form of cyber attack that specifically seeks to cause inflated usage bills through the abuse of serverless functions. This attack is called Denial of Wallet (DoW). This thesis presents the first in depth investigation in academia on DoW, comprising the formal definition of the attack, theorised attack vectors, a means of safely recreating attacks for research purposes via synthetic data generation, and a novel detection strategy utilising image classification that yields a detection accuracy of 97.98%

Table of Contents

List of Figures

List of Tables

Nomenclature

Acronyms / Abbreviations

API Application Programming Interfaces

AWS Amazon Web Services

CDN Content Delivery Network

CGI Common Gateway Interface

CIA Criminal account Inference Algorithm

CNN Convolutional Neural Network

DDoS Distributed Denial of Service

DNS Domain Name System

DoS Denial of Service

DoW Denial of Wallet

DoWTS Denial of Wallet Test Simulator

EDoS Economic Denial of Sustainability

EMD Entropy Minimization Discretization

FaaS Function-as-a-Service

GAN Generative Adversarial Networks

HTTP Hyper Text Transfer Protocol

IaaS Infrastructure-as-a-Service

ICMP Internet Control Message Protocol

ILSVRC ImageNet Large-Scale Visual Recognition Challenge

IoT Internet-of-Things

IP Internet Protocol

IPS Intrusion Prevention System

IRC Internet Relay Chat

ISP Internet Service Provider

JSON JavaScript Object Notation

KSTest Kolmogorov Smirnov Test

LBP Local Binary Patterns

LOIC Low Orbit Ion Cannon

MCMC Markov Chain Monte Carlo

NIDS Network Intrusion Detection System

NoOps No Operations

OCR Optical Character Recognition

OWASP Open Worldwide Application Security Project

PaaS Platform-as-a-Service

PPC Pay-Per-Click

REST REpresentational State Transfer

rph requests per hour

*S*3 Simple Storage Service

SaaS Software-as-a-Service

SAE Stacked Auto-Encoder

SMB Server Message Block

SPE Serverless Platform Emulator

sPoW self-verifying Proof of Work

SSH Secure Shell

SYN SYNchronize packet

TCP Transmission Control Protocol

UDP User Datagram Protocol

VAE Variational Autoencoders

VF Virtual Firewall

VM Virtual Machine

WAF Web Application Firewall

CHAPTER 1

Introduction

Serverless computing is a method of providing back-end services on an as-used basis. A serverless provider allows users to write and deploy code without the hassle of worrying about the underlying infrastructure. A company that gets back-end services from a serverless vendor is charged based on their computation and do not have to reserve and pay for a fixed amount of bandwidth or number of servers, as the service is auto-scaling. Since the launch of AWS Lambda circa 2014, serverless computing has seen a steady rise of uptake by businesses as they migrate their monolithic applications to the cloud. Some benefits of this move are the increased load capacity and ease of scaling, as the developer is not responsible for designing and maintenance of the back-end resources. The serverless computing market is predicted to reach a value of $36.84 billion by 2028 [3].

1.1 Motivation

Serverless computing is by no means a perfect model. With its unique combination of *pay-as-you-go* computing and its capability for massive scaling, it allows for the execution of a potential attack, Denial of Wallet (DoW), which is *the intentional mass and continual invocation of serverless functions, resulting in financial*

exhaustion of the victim in the form of inflated usage bills. This attack is currently considered theoretical as there have been no documented instances of targeted DoW *in the wild.* However, this is not a convincing indicator that it has not been carried out. Given that the only victim of DoW is the bill payer of the application, end users would not be directly affected by the attack. As such, there would be no need for a business to declare that they were attacked, keeping it secret. Also, as will be discussed throughout this research, there are potential forms of this attack that may never be detected, as they may not arouse suspicion.

The current state of serverless security does not openly address the issue of DoW with specific tools. Instead there is a reliance on existing mitigation strategies for other attacks and placing the onus on developers to take the security of their functions into their own hands. We believe this is not an appropriate approach to such a looming threat. It is important to take advantage of this opportunity to conduct pre-emptive research on this attack, as it is a rare occasion where the defence can be a step ahead of the attackers in the arena of cybersecurity. The unique attack vectors of DoW require the development of novel research tools and approaches in order to understand and create the first formal analysis of this looming threat and answer the question of whether it can be accurately detected and defeated.

1.2 Research Questions

This thesis aims to answer the following research questions:

1. What makes DoW unique to other attacks and how does this influence the potential design of its attack patterns? (RQ1)

2. How can a lack of historical data be overcome in the preemptive investigation of this attack, where traditional sandbox testing is not viable? (RQ2)

3. Can an attack that so closely mimics regular traffic be detected and mitigated against? (RQ3)

4. What is the potential direction DoW may take in the hands of malicious actors for increased damage and success of attack? (RQ4)

1.3 Hypotheses

From the research questions outlined above, this thesis expects to demonstrate that:

1. DoW is a threat to serverless computing that warrants this early investigation where there has not currently been a disclosed instance of the attack. As a result, initial theorised attack patterns and intelligent means of detection are required in order to be a step ahead of potential attacks.

2. In a position of no available data on which to conduct research, synthetic data generation is a viable source of material for the continued development of DoW attack classifiers.

3. By devising a novel means of representing request traffic, detection of attacks that purposefully mimic normal traffic is possible by training classification algorithms on said representations.

1.4 Contributions

The main contributions of this research are:

- The first formal analysis and definition of DoW in academic literature.

- Creation of a safe means of testing DoW detection approaches without incurring financial damage, Denial of Wallet Test Simulator (DoWTS). Through the use of synthetic data generation, any number of DoW scenarios can be created for further research, thus closing the knowledge gap where there is a total lack of historical data demonstrating DoW.

- Development of a novel means of representing large volumes of request traffic to function triggers. These representations were then used to train an image classifier for use in an early warning system for DoW detection that achieved more than 97% accuracy.

1.5 Publications

The main contents of this thesis have been published in journals and conference proceedings. The list of publications is found below.

1.5.1 Journal Papers

1. **Daniel Kelly**, Frank G. Glavin and Enda Barrett
   "Denial of Wallet - Defining a Looming Threat to Serverless Computing,"
   Journal of Information Security and Applications 60 (2021): 102843
   Impact factor - 4.96 *Citations* - 14

2. **Daniel Kelly**, Frank G. Glavin and Enda Barrett
   "DoWTS – Denial-of-Wallet Test Simulator: Synthetic Data Generation for Preemptive Defence"
   Journal of Intelligent Information Systems 60, no. 2 (2023): 325-348
   Impact factor - 2.9 *Citations* - 1

1.5.2 Conference Proceedings

1. **Daniel Kelly**, Frank G. Glavin and Enda Barrett
   "Serverless Computing: Behind the Scenes of Major Platforms"
   2020 IEEE 13th International Conference on Cloud Computing (CLOUD) pp. 304-312. IEEE, 2020
   Citations - 24

1.5.3 Poster Presentations

1. **Daniel Kelly**, Frank G Glavin and Enda Barrett
   "Poster: Denial of Wallet Preemptive Defence-Attack Simulation and Vulnerability Scouting"
   43rd IEEE Symposium on Security and Privacy, IEEE S&P 2022

1.5.4 Under Review

1. **Daniel Kelly**, Frank G Glavin and Enda Barrett (2023)
   "DoWNet - Classification of Denial of Wallet Attacks on Serverless Application Traffic"

1.6 Thesis Overview

This thesis is made up of seven chapters and an appendix. Chapters 3, 4, 5 and 6 contain our peer reviewed articles that detail the evolution of this work and major contributions to the field.

The content of each chapter is as follows:

- Chapter 1: We highlight the motivation and contributions of this research.
- Chapter 2: We provide a background and review of the literature on related work that gives context to the research in this thesis.
- Chapter 3: We present the first published definition of the DoW attack. We discuss the potential threat to serverless computing and highlight a number of theoretical use cases of its unique attack capabilities. The content of this chapter is in part taken from "Denial of Wallet - Defining a Looming Threat to Serverless Computing". We extend the discussion on DoW attacks beyond what is discussed in our previous publications. Namely, a deeper discussion on attack strategies and difficulties with accurate detection and meaningful mitigation.
- Chapter 4: We perform a detailed evaluation of the underlying topology of major serverless platforms that introduces the threat of the DoW attack. The content of this chapter is taken from "Serverless Computing: Behind the Scenes of Major Platforms"
- Chapter 5: We develop a synthetic data generator for the safe testing of DoW detection strategies. This chapter looks at synthetic data use in cybersecurity and demonstrates the Denial of Wallet Test Simulator (DoWTS). Also, we design a safe isolated testbed for testing attack and mitigation strategies. This testbed replicates a commercial platform for more realistic protection

system evaluation. The content of this chapter is taken from “Denial of Wallet - Defining a Looming Threat to Serverless Computing” and “DoWTS – Denial-of-Wallet Test Simulator: Synthetic Data Generation for Preemptive Defence”.

- Chapter 6: We create an early warning detection system utilising image classification. We devise a novel approach to web application traffic visualisation that allows for the use of image recognition techniques to detect DoW attacks. The content of this chapter is taken from “DoWNet - Classification of Denial of Wallet Attacks on Serverless Application Traffic”.
- Chapter 7: We give the main conclusions of the research presented and we discuss the limitations and potential for future work.

CHAPTER 2

Background and Literature Review

This chapter presents the relevant background material required to understand the contributions presented in this thesis. The topics covered address cloud computing as a whole and, more specifically, serverless computing, synthetic data generation, related cyber threats, machine learning, and computer vision.

2.1 Cloud Computing

2.1.1 Background

Cloud computing is a technology that enables the delivery of computing services over the Internet. Instead of running applications or storing data on local devices, cloud computing provides access to computing resources such as servers, storage, and software applications remotely. This technology has become an essential tool for businesses of all sizes.

The concept of cloud computing dates back to the 1960s when the idea of time-sharing was introduced to the IBM 7090 machine [4].

> *"Computation may someday be organized as a public utility, just as the telephone system is a public utility. We can envisage computing service companies whose subscribers are connected to them by telephone lines.*

> *Each subscriber needs to pay only for the capacity that he actually uses, but he has access to all programming languages characteristic of a very large system*" - John McCarthy [5]

Cloud computing became a reality with the development of virtualisation technology and the availability of high-speed Internet in the 2000s. Amazon Web Services (AWS) launched in 2006 and provided the first widely available cloud computing platform. The popularity of cloud computing grew rapidly in the 2010s, as more businesses and organisations began to adopt cloud computing for their IT infrastructure. In addition to AWS, Google Cloud, and Microsoft Azure, other major cloud computing providers such as IBM, Oracle, and Alibaba Cloud also emerged during this period [6].

The types of cloud computing service models offered by providers are [7]:

1. **Infrastructure-as-a-Service (IaaS)** - is a type of cloud computing service model where users can rent or provision virtualised computing resources such as servers, storage or networking, on a pay-per-use basis from a cloud provider. In IaaS, the cloud provider is responsible for managing the infrastructure, including hardware, networking, and storage, while the user is responsible for managing the operating system, applications, and data. This allows users to focus on their core business operations rather than managing hardware and infrastructure. IaaS offers several advantages, including scalability, cost-effectiveness, and flexibility. Users can easily scale their infrastructure up or down based on their needs, pay only for the resources they use, and have the flexibility to choose the operating system, applications, and tools that best meet their requirements.

2. **Platform-as-a-Service (PaaS)** - is a cloud computing service model where users can rent or provision a platform for developing, running, and managing applications. As in IaaS, the cloud provider is responsible for managing the infrastructure. Additionally, the platform, including the operating system, middleware, and runtime environment are also managed. Users are responsible for developing and deploying their applications on the platform. PaaS providers offer services such as application hosting and database management, and provide users with APIs and tools to easily manage their

applications. Some popular examples of PaaS providers include Heroku, Microsoft Azure, and Google App Engine. PaaS is often used by developers to quickly build and deploy web applications, mobile applications, and other types of software without having to worry about the underlying infrastructure.

3. **Software-as-a-Service (SaaS)** - is another cloud computing model where users can access and use software applications hosted by a cloud provider. In SaaS, the cloud provider manages all aspects pertaining to maintenance of the platform and software, and users access the software application through some interface, such as a web browser. This eliminates the need to download and maintain software updates. SaaS providers offer a range of software applications such as productivity tools, customer relationship management software, and enterprise resource planning software, and provide users with Application Programming Interfaces (API) and integration options to easily integrate with other software applications. Some popular examples of SaaS providers include Salesforce, Microsoft Office 365, and Google Workspace.

There is a fourth service model, *Function-as-a-Service* (FaaS) also known as Serverless Computing, which is the domain of this research and is described in detail in this section.

There are great benefits to cloud computing over *on-premises* resource hosting, the most notable of which being its cost effectiveness. Cloud computing eliminates the need to invest in expensive infrastructure and allows businesses to pay for computing resources on an as-needed basis. Another major benefit to resource hosting in the cloud is the capacity for scalability. Cloud computing allows businesses to scale their computing resources up or down as needed without worrying about physical infrastructure. While the upwards scaling ability is important for applications expecting large bursts in traffic, the ability to down scale is equally useful. Cloud computing allows for this without the need to manage unused hardware, as would be the case if managing resources *on-premises*. Additional benefits include increased accessibility as users can access their data and applications from anywhere with an internet connection, integrated baseline security measures, and continued access to the latest innovation in cutting-edge technolo-

gies such as machine learning and AI that would be challenging to implement *on-premises*.

However, despite the numerous benefits, there are challenges that must be met when utilising cloud computing. While security measures are offered by each platform, businesses need to ensure that their data and applications are adequately secured. Businesses also must ensure compliance in their use of cloud computing, notably data storage and access. Finally, it is worth a business's time to plan how they can avoid vendor lock-in to ensure that they can easily switch to another cloud provider if needed.

Serverless Computing has emerged as a powerful paradigm and service model for application development. Contrary to what the name implies, serverless computing does actually use servers. However, the management of these servers is handled by the cloud provider. The concept of serverless computing can be traced back to the early days of the Internet, when Common Gateway Interface (CGI) scripts were used to dynamically generate HTML pages [8]. The concept of serverless computing began to take shape in the 2010s. The term *serverless* was coined in 2012 by Ken Fromm [9]. AWS Lambda launched in 2014, which allowed developers to run code in response to events without having to provision or manage servers. This gradual rise in popularity can be seen in Google trends data on searches for *serverless computing* (Figure 2.1). Serverless computing has become an increasingly popular way for developers to build and deploy applications. With serverless computing, developers utilise back-end services from cloud providers on a *pay-as-you-go* basis, meaning they pay only for the services used and never pay for idle capacity. The cloud provider spins up and provisions the required computing resources on demand when the code executes, and spins them back down when execution stops, termed *scaling to zero*. Serverless computing is attractive to developers in that it reduces the effort allocated to server management, security, and scalability. With serverless, routine tasks such as managing the operating system and file system, security patches, load balancing, capacity management, scaling, logging, and monitoring are all offloaded to a cloud services provider.

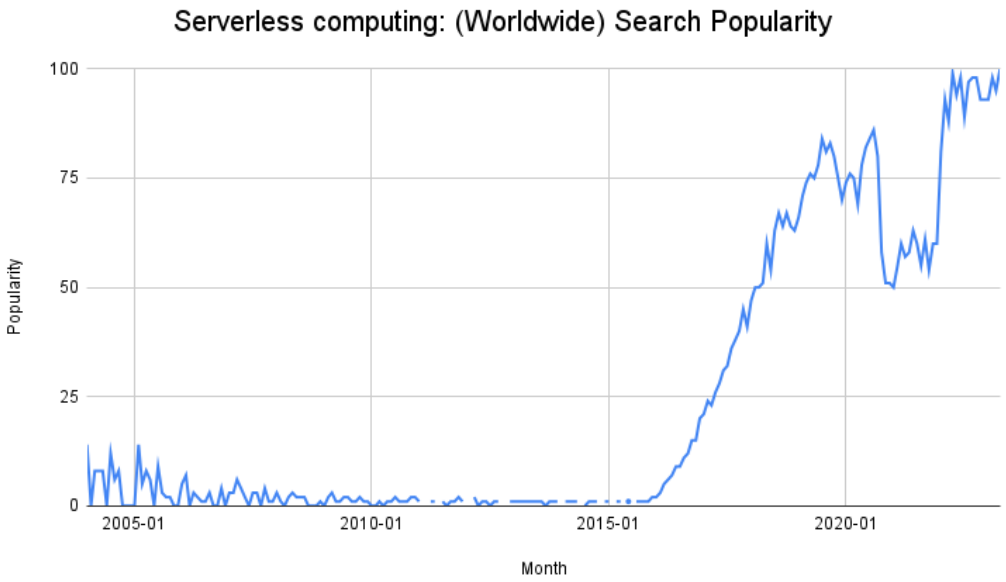

Fig. 2.1 Emergence of popularity of serverless computing. Numbers represent search interest relative to the highest point on the chart for the given region and time. A value of 100 is the peak popularity for the term. A value of 50 means that the term is half as popular. A score of 0 means that there was not enough data for this term. "Data source: Google Trends (https://www.google.com/trends)."

2.1.2 Usage of Serverless Computing

Serverless differs from other cloud computing models in that the cloud provider is responsible for managing both the cloud infrastructure and the scaling of applications. Serverless apps are deployed in containers that automatically launch on demand when called.

On applications that utilise an IaaS cloud computing model, developers provision all components necessary to run the application in an *always on* system (in order for there to be functionality, the components must always be running, waiting for interaction). It is the developer's responsibility to monitor traffic loads and scale up server capacity during times of high demand and to scale down when that capacity is no longer needed.

With serverless computing, by contrast, apps are launched only as needed. When an event triggers a function to run, the cloud provider dynamically allo-

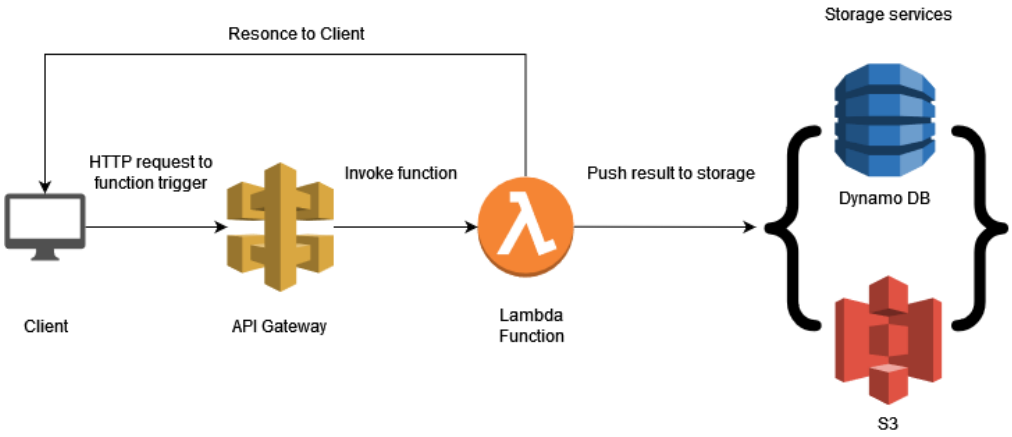

Fig. 2.2 Typical life-cycle of serverless function utilisation (example of AWS products)

cates resources for that code. The developer will only be billed for the resources consumed during the execution time of the function [10].

Applications are developed for each desired process and the event that invokes it via constructs called *functions*. Serverless function platforms provide the infrastructure to deploy code for execution across their cloud and define the event processing logic that prompts the functions to run using the model: event, trigger, and action. Serverless computing promotes the idea that application development is abstracted from the underlying infrastructure and that there is no need for a dedicated team to manage software in-house, embracing the term “NoOps” (no operations)[11]. Serverless computing abstracts back-end management from users, allowing only minimal access to some basic parameters such as function memory allocation and function runtime timeout.

The billing of serverless function execution varies slightly by cloud provider. In general, they utilise the following system:

Charges are based on the number of requests to the functions and the time it takes for the code to execute. The cloud provider registers a request each time it starts to execute in response to an event notification or invoke call, including test invokes from the console. Charges are for the total number of requests across all the functions. Duration is calculated from the time the code begins executing until it returns or otherwise terminates, rounded up to the nearest 1ms. The price depends on the amount of memory that is allocated to the function. Typical rates

would be $0.00001667 for every GB-second [12]. Each provider will also include a free usage tier, usually around 1 million requests and 400,000 GB-seconds of usage.

2.1.3 Pros and Cons of Serverless Computing

Serverless computing is seeing continued growth as more developers are choosing to utilise it in new applications and existing companies are migrating their legacy software to the cloud. The advantages and disadvantages are reminiscent of those discussed on cloud computing above. However, there are more examples specific to serverless as follows. The advantages include the following:

- **Cost-effectiveness** - Users and developers pay only for the time when code runs on a serverless compute platform. They do not pay for idle virtual machines (VM) as serverless containers can scale-to-zero.
- **Ease of deployment** - Developers can deploy apps in hours or days rather than weeks or months. Time can be spent writing and developing apps instead of dealing with servers and runtimes.
- **Autoscaling** - Cloud providers handle scaling up or spinning down resources or containers when the code is not running.

The disadvantages of serverless computing include the following:

- **Vendor lock-in** - Switching cloud providers might be difficult because the way serverless services are delivered can vary from one vendor to another. There is often a heavy reliance on other internal services on the cloud platform.
- **Inefficient for long-running apps** - Sometimes using long-running tasks can cost much more than running a workload on a VM or dedicated server.
- **Latency** - There's a delay in the time it takes for a scalable serverless platform to handle a function for the first time, often known as a cold start.
- **Debugging is more difficult** - Because a serverless instance creates a new version of itself each time it spins up, it is hard to acquire the data needed to debug and fix a serverless function.

2.1.4 Use cases for Serverless

Given its unique combination of attributes and benefits, serverless architecture is well suited for use cases around microservices, mobile backends, and data and event stream processing.

Event-triggered computing - For scenarios that involve numerous devices accessing various file types, such as mobile phones and PCs uploading videos, text files, and images.

Serverless and microservices - Supporting microservice architectures is one of the most common uses of serverless computing. The microservices model is focused on creating small services that do a single job and communicate with each other using APIs. Although developers can use containers or PaaS to build and operate microservices, they can also use serverless computing because of its inherent and automatic scaling, rapid provisioning, and pricing model that only charges for the capacity used.

API backends - Serverless computing makes it easier to build RESTful APIs that developers can scale up on demand. Any action (or function) on a serverless platform can be turned into an HTTP endpoint ready to be consumed by web clients. When enabled for web, these actions are called web actions. These can be combined into a full-featured API with an API gateway that brings additional security, OAuth support, rate limiting, and custom domain support.

Data processing - Serverless is well suited to work with structured text, audio, image, and video data, around tasks such as data enrichment, transformation, validation, cleansing, PDF processing, audio normalisation, image processing (rotation, sharpening, noise reduction, thumbnail generation), optical character recognition (OCR), and video transcoding.

In summary, serverless computing is rapidly increasing in popularity and uptake. There are many benefits to its highly obfuscated back-end management, especially for developers looking to quickly integrate business logic into their services. FaaS is the latest addition to the *as-a-Service* family and has been widely adopted by industry for its ease of scaling capabilities and cost-effectiveness [3].

2.2 Synthetic Data Generation

2.2.1 Background

Synthetic data generation is the process of creating artificial data that mimics the characteristics of real data. This is often done to overcome the limitations of real-world data, where perhaps there are privacy concerns or data scarcity. Synthetic data generation has a wide range of applications across many fields beyond cybersecurity. Some common use cases of synthetic data include:

- **Data augmentation** - is a technique commonly used in machine learning to increase the amount of data available for training models. It involves creating modified copies of existing data or generating new synthetic data, which can help to reduce overfitting by increasing regularisation. This technique can be particularly useful in addressing under-sampling within a dataset, such as when a specific subgroup of the population is underrepresented i.e. an imbalanced dataset. By adding synthetic data to boost that subgroup, the model's performance may improve, resulting in a more inclusive and fair model overall. It is important to exercise caution when interpreting the results of data augmentation and to ensure that the training and evaluation datasets are kept completely separate to avoid data leakage [13].

- **Prevention of privacy attacks on machine learning models** - The use of machine learning models can pose a risk to data security through privacy attacks aimed at uncovering the data used for training. With the increasing prevalence of white box and black box attacks, it has become easier to access and retrieve training data, potentially leading to the leakage of personal information. However, one way to mitigate this risk is by using synthetic data to develop machine learning models. By constructing models using synthetic data, the risk of data leakage is eliminated, as any training data recovered by attackers would only be synthetic variants. This not only strengthens security, but also enables models to be shared more widely in the scientific community, optimising learning and utility in this field. This use case highlights the potential of synthetic data to enhance security, facilitate sharing, and reduce the risk of disclosing real data that may contain personal information [13].

While synthetic data generation has many potential benefits, there are also several challenges and considerations to keep in mind. Bias is a major issue in all datasets, regardless of whether it is synthetic or not. This can be because of inherent bias in the creator of the dataset, e.g. an image dataset of people containing a disproportionate number of white males. Synthetic data generation can compound the issue, as it will be trained to mimic said biases, thus creating an unfair representation of reality by amplifying those biases. Also, such a generator may be too rigid in its produced data, whereas real data is dynamic and evolving. This leads to a greater concern of AI models being fed synthetic data becoming a closed loop system where it will make predictions based on this data and upon further training begin to diverge further from reality [14].

2.2.2 Methods of Data Synthesis

There are several techniques for generating synthetic data, each with its own strengths and weaknesses. Some of the most common techniques include:

Generative Adversarial Networks (GAN) are a type of deep learning architecture that consists of two neural networks: a *generator* and a *discriminator*. The generator network is trained to create synthetic data, such as images, audio, or text, similar to the training data. During training, the generator generates synthetic data, and the discriminator tries to distinguish between the synthetic and real data. The feedback from the discriminator is then used to update the generator, so it can create more realistic synthetic data. This process continues until the generator can generate synthetic data that is indistinguishable from the real data, at which point the discriminator is no longer able to differentiate between the two. GANs can generate new data that have characteristics similar to those of the training data. However, they can also be challenging to train due to their instability and the potential for mode collapse, where the generator produces limited varieties of data [15]. GANs have seen successful use in supplementing medical imaging data for training of various medical condition detection systems [16, 17]. They are especially useful where it is difficult to obtain enough real data given the nature in which such images must be obtained (finding a patient with the ailment who will consent to their images being added to the dataset). However, even in applications where there are less ethical considerations for data collection, GANs

have shown their effectiveness in synthesising not only images for applications such as vehicle registration plate recognition [18] and infrared target detection [19] but also tabular data [20].

Variational Autoencoders (VAE) are a type of generative model that can learn to generate new data similar to the training data. VAEs are a type of neural network that can learn to represent high-dimensional data in a lower-dimensional latent space. The VAE architecture comprises two parts: an *encoder* and a *decoder*. The encoder takes in an input data and maps it to a latent space representation, and the decoder takes the latent space representation and maps it back to the original input data space. The VAE learns to reconstruct the input data as closely as possible while also learning to create diverse samples from the latent space [21]. VAEs have demonstrated their ability in generating synthetic images, such as faces [22, 23] and text characters [24]. However, they have also shown capability in niche domains, synthesising network traffic [25] and chemical molecules [26].

Markov Chain Monte Carlo (MCMC) methods are a class of algorithms used in statistical analysis and machine learning to generate samples from a complex distribution. MCMC methods allow for the approximation of the behaviour of a complex system by simulating a Markov chain of samples that converge to the desired distribution. A Markov chain is constructed in which each sample is generated from the previous sample by a transition probability that depends only on the current state. The next sample in the chain is generated on the basis of the current sample, and the process repeats for a large number of iterations. As the number of iterations increases, the samples in the chain become more representative of the desired distribution. MCMC methods can be computationally expensive and it can be challenging to determine when the Markov chain has converged to the desired distribution. There are several popular MCMC methods, including Metropolis-Hastings and Gibbs sampling [27].

However, within the scope of this research, we focus on the use of *heuristic-based* synthetic data generation.

Heuristic-based synthetic data generation is a method that uses rules or heuristics to create data that resembles real-world data. This approach is often used when there is limited recorded data available or when the available data is not

representative of the full range of possible scenarios. A set of heuristics is used to generate data that matches the statistical properties and distribution or correlation structure of existing data.

One heuristic-based approach is to use a data model to generate data that is statistically similar to the real-world data. The model is evaluated on a number of statistical measurements such as Wasserstein Distance, Jensen Shannon Divergence and Kolmogorov Smirnov Test Statistic. Another approach is to use expert knowledge to create heuristics that describe the underlying structure of the data. For example, an expert in a particular field may be able to identify patterns or relationships in the data that can be used to generate synthetic data that matches the real-world data.

Heuristic-based synthetic data generation has advantages over the previously discussed methods of generating synthetic data. It can be faster and more efficient than other methods, as it does not require as much computational power or resources. It can also be more interpretable, as the heuristics used to generate the data can be easily understood and explained. However, heuristic-based methods may not always be as accurate as other methods, particularly when heuristics do not fully capture the complexity of the real-world data.

2.2.3 Synthetic Data in Cybersecurity

Synthetic data generation is becoming increasingly important in the field of cybersecurity. Large volumes of data are required in order to identify and respond to cyber threats, but there may be no access to real-world data due to privacy concerns or other constraints as mentioned above.

One way synthetic data generation is used in cybersecurity is to create realistic datasets for training and testing machine learning algorithms by generating datasets that mimic real-world cyber attacks [28, 29]. Algorithms can then be trained to recognise and respond to these attacks more effectively. This can help improve the accuracy and reliability of cybersecurity systems.

Synthetic data generation can also be used to protect sensitive data in cybersecurity. By generating synthetic data that is statistically similar to real-world data, researchers can create datasets that can be used for testing and development without risking the exposure of sensitive information, such as location information, payment details, purchase or search history, etc.

Another application of synthetic data generation in cybersecurity is to create synthetic network traffic [30, 31]. By generating synthetic network traffic that resembles real-world traffic patterns, the performance of cybersecurity systems can be tested and potential vulnerabilities or weaknesses can be identified. This particular application is especially pertinent to this research, as we had to overcome a lack of historical network traffic that demonstrates the DoW attack.

Synthesising web application traffic refers to the process of creating realistic web traffic patterns to simulate user behaviour on a web application. This can be useful for testing the performance and security of web applications, as well as for generating synthetic datasets for training machine learning models.

There are a number of techniques and tools available for synthesising web application traffic. In industry, it is common to use a traffic generator tool, such as Apache JMeter[1] or Selenium WebDriver [2], to simulate user behaviour by sending HTTP requests to the web application. The tool can be configured to generate requests that mimic the behaviour of real users, such as clicking links, filling out forms, and submitting requests. For attack traffic, adversarial software such as Low Orbit Ion Cannon (LOIC) [32] may be used to send HTTP floods.

Beyond the tools employed by those working in cybersecurity, any of the methods of generating synthetic data mentioned previously may be used in conjunction with a simple web client such as cURL to handle the actual sending of requests. The synthetic data generators can recreate the timings of regular traffic which are then fed to the web client for transport.

Examples of synthetic network traffic generation in the literature include:

Cordero *et al.* [30] did create a tool to allow for standardised replication of network intrusion detection research by generating synthetic attack traffic by replicating background traffic properties. They demonstrate the ability to inject attack traffic using their tool rather than requiring the release of specific datasets for that attack.

Xu *et al.* [31] did create a tool to generate network traffic. They run an exhaustive suite of tests for validating the data which includes likelihood testing; negative log likelihood and performance in machine learning tasks.

[1] https://jmeter.apache.org/
[2] https://www.selenium.dev/documentation/webdriver/

However, these approaches are currently best suited to non-serverless application architectures and real-time (1 to 1) data generation i.e. it takes an hour to create an hour's worth of network traffic. For our research we require large amounts of archived network traffic data. Our solution to this is discussed in Chapter 5.

2.3 Understanding the Threat Domain

2.3.1 Background

In this research we are investigating an attack that, although it has not been publicly observed specifically, shares characteristics with other related attacks. As such, it is important to be aware of these various related threats in order to better theorise attack and defence strategies.

Denial of Service

Denial of Service (DoS) is a type of cyber attack that involves disrupting or interrupting the normal functioning of a website, network, or server by overwhelming it with traffic or requests. The aim of a DoS attack is to make the targeted system unavailable to its users, causing inconvenience, financial losses, or reputational damage to the targeted organisation.

There are several types of DoS attacks, each with its own characteristics and techniques. Examples of DoS attack types are [33]:

- **Data Flooding Attacks** - These volumetric attacks flood the target network or server with a large number of requests or traffic, overwhelming its capacity to handle them. This type of attack is often carried out using botnets, which are networks of compromised devices controlled by an attacker.

- **Protocol Attacks** - These attacks exploit vulnerabilities in network protocols, such as TCP/IP, DNS, and ICMP, to disrupt the communication between the target system and its clients. Protocol attacks can be more sophisticated and difficult to detect than volumetric attacks.

- **Application-layer Attacks** - These attacks target specific applications or services running on the target system, such as web servers, email servers,

or databases. Application-layer attacks can be more targeted and effective than other types of attacks, as they exploit vulnerabilities in the software or configuration of the target application.

DoS attacks can have impact beyond the immediate taking down of a service. A DoS attack on a government website can disrupt the delivery of essential services to citizens, while on a financial institution can cause knock on financial losses. DoS attacks on healthcare organisations can compromise patient data and disrupt critical medical services, while on e-commerce websites can result in lost revenue and damage to customer trust.

Economic Denial of Sustainability

Economic Denial of Sustainability (EDoS) is a type of cyber attack that targets the economic resources of a system, with the aim of disrupting its functioning and causing economic damage as a result of mass scaling of the service [34]. EDoS attacks can be carried out in various ways, but generally they exploit vulnerabilities in the pricing or resource allocation models of a system to create artificial scarcity or increase costs, which can make the system unsustainable.

An EDoS attack on a cloud computing service provider could involve a malicious user or group that consumes a large amount of resources without paying for them, creating congestion, and driving up costs for legitimate users. This can lead to a situation where the provider is unable to sustain its operations or maintain quality of service, leading to service disruptions and financial losses.

EDoS attacks can have significant impacts on the economic viability and sustainability of a system, as they can reduce revenue, create mistrust among users, and discourage new investment. EDoS attacks can also increase costs and make the system less accessible to smaller users, further exacerbating the economic damage. As Chapter 3 will discuss, EDoS is the precursor to DoW, with DoW being a term that is more commonly used and arguably more specific, as this thesis will demonstrate.

Click Fraud

Perhaps the most prevalent form of cyber attack that specifically targets the finances of an application owner is click fraud. It is an issue that has plagued

companies with an online advertising presence long before the introduction of FaaS or the emergence of EDoS. It is a type of fraudulent activity that involves repeatedly clicking on online ads, often with the aim of generating revenue for the fraudster or causing financial harm to the advertiser.

Click fraud involves artificially inflating the number of clicks on online ads, typically through the use of automated scripts or bots. The goal of click fraud is often to generate revenue for the fraudster, as on-line ads are often sold on a pay-per-click (PPC) basis, meaning that advertisers pay each time their ads are clicked. By clicking on ads repeatedly, fraudsters can drain the budgets of advertisers and divert traffic away from legitimate sites.

Click fraud can also be used as a form of sabotage or retaliation, as a competitor or disgruntled individual can click on an advertiser's ads repeatedly in order to exhaust their budget and reduce the effectiveness of their campaigns.

These two categories of click fraud are known as [35]:

- **Inflationary Click Fraud** - Third parties may click on advertisements hosted on their service in order to inflate their revenues.
- **Competitive Click Fraud** - Rivals fraudulently click on their competitor's advertisements in order to drive up their advertising costs, thus exhausting their budget.

Click fraud can have a significant impact on the advertising industry, as it can reduce the effectiveness of advertising campaigns and drain advertising budgets. Click fraud can also lead to mistrust and decreased confidence in online advertising, as advertisers may be reluctant to invest in ad campaigns if they believe that their budgets will be wasted on fraudulent clicks. Click fraud was estimated to cost $6.5 billion dollars globally despite 80% of brands (participating in the study), deploying some form of fraud countermeasure [36].

2.3.2 Related Threat Mitigation

Since occurrences of DoW are limited to a only handful of examples spoken about online, we are in a rare position of being ahead of the attackers. We can preemptively develop mitigation strategies rather than reacting to an attack when it occurs. Application layer DoS attacks are the strongest contender for the starting

point of effective DoW attacks, as well as Economic Denial of Sustainability. The former could be re-purposed for DoW use, potentially leaving APIs, modules and libraries vulnerable to such modified attacks. The latter being the closest analogue to DoW that has been researched previously. Therefore, it is advantageous to be aware of mitigation strategies for those attacks and more.

Denial of Service Mitigation

Detecting and mitigating DoS attacks require a combination of technical, organisational, and policy measures. Some common techniques and best-practices used to prevent or mitigate DoS attacks are as follows.

1. **Network-level Protection** - This includes measures such as firewalls, intrusion detection systems, and traffic filtering to block malicious traffic and prevent unauthorised access to the network.

2. **Application-level Protection** - This includes measures such as implementing secure coding practices, using secure protocols, and regularly patching software vulnerabilities to prevent application-layer attacks.

3. **Cloud-based Protection** - Cloud-based protection services, such as Content Delivery Networks (CDN) and external DoS mitigation services such as scrubbing servers, absorb and filter traffic before it reaches the target system, reducing the impact of volumetric attacks.

Inspiration for DoW mitigation may come from research carried out by DoS mitigation. Barna *et al.* [37] proposed a system for DoS mitigation that dynamically changes the rules on a firewall to minimise false positive detections and then pass suspicious activity to a CAPTCHA challenge. The application layer was the area of the web application being monitored. This makes this mitigation system relevant to DoW research, as DoW attacks are conducted on the application layer via API triggers. Their experimental setup of a mock application that they then attacked with real DoS tools served as inspiration for the experimental setup devised in Chapter 5.

These initial insights into application layer DoS mitigation are a useful starting point. However, this research will aim to employ more sophisticated means of

detection with the use of machine learning. Machine learning approaches to DoS detection are discussed in Section 2.4.

Economic Denial of Sustainability

In general, since EDoS is a financial exhaustion attack, improved policies and diligence in billing and resource management will prove to be the most effective defence. Ensuring that the pricing and resource allocation models of a system are designed to resist and mitigate EDoS attacks is a critical step in preventing them from occurring in the first place.

Implementing technical measures such as rate limiting, resource allocation policies, and congestion control can help to reduce the impact of EDoS attacks by limiting the amount of resources that can be consumed by malicious actors. Also, regularly monitoring the system for signs of EDoS attacks, such as unusual resource consumption, large amounts of traffic, and cost spikes, can help to detect and respond to attacks early, reducing their impact.

As EDoS is the precursor to DoW, it is beneficial to look towards existing research on specific mitigation approaches.

Kumar *et al.* [38] propose a model that operates in two modes depending on the load imposed on the server; *Normal Mode* and *Suspect Mode*. When the resource consumption threshold is reached, the service provider activates *Suspect Mode* in anticipation of a Distributed Denial of Service (DDoS) attack. In this mode, the client machine must now solve a mathematical puzzle when a request is made.

Khor *et al.* [39] also propose a puzzle-based mitigation approach with self-verifying Proof of Work (sPoW). sPoW was designed in such a way that it could be implemented into existing cloud technology. It returns a puzzle with an encryption key when a client tries to connect to a server. The client communicates with the server through plugins embedded in a static webpage. The server plugin handles puzzle generation by creating a temporary encryption channel and sends a puzzle to the client to solve. If successful, the connection can proceed; otherwise, it will return a more difficult puzzle in order to re-establish the connection.

Idziorek *et al.* [40] proposed two detection methodologies: Zipf's Law and Entropy detection. Zipf's law is used to detect anomalous patterns in incoming traffic,

whereby anomalous traffic will not fit the frequency distributions of *ranked data* being the expected traffic. Entropy-based detection analyses individual user traffic and calculates the entropy of session lengths over a specific time.

Sqalli *et al.* [41] devised *EDoS Shield*. It consists of virtual firewalls (VF) and verifier nodes (V-nodes). Client requests are sent to the VF which forwards the request to a V-node. The V-node sends a graphical Turing test to the client, such as Completely Automated Public Turing test to tell Computers and Humans Apart (CAPTCHA). If successful, the client's IP address is updated to the whitelist in the VF. Failing the test leads to the source IP address being added to the blacklist with subsequent packet requests being dropped.

Click Fraud

Click fraud schemes are either executed by low-wage workers from developing countries or large botnets [42]. These botnets execute ever increasingly sophisticated algorithms that mimic the behaviour of legitimate users. In 2016-2017, more than 76% of fraud detections were found to be from machines that hosted both a bot and a human [43]. This makes the task of detecting such illegitimate traffic much more difficult. Detection approaches for click fraud include:

- **Implementing Click Fraud Detection Systems** - Advertisers can use specialised software tools and algorithms to detect click fraud and identify suspicious activity. These systems identify patterns of fraudulent behaviour and flag suspicious activity such as mouse movement [36] for review.
- **Using IP Address Filtering** - Advertisers and can also use IP address filtering to block traffic from known sources of click fraud, defining geographical areas of suspicion for incoming traffic [42].
- **Ad Fraud Policies** - Advertisers can establish clear policies and guidelines around click fraud, including what constitutes fraudulent activity and how it will be dealt with. Ad networks can also establish policies and penalties for publishers who engage in click fraud or other fraudulent activities.

2.3.3 Serverless Attack Surface

Serverless functions can be invoked by various triggers. By far the most common method is via API endpoints. Some platforms such as Google Cloud will create a URL endpoint for a function once it is deployed, whereas others such as AWS requires to link functions to an API Gateway. The result is a means of executing functions via a request to the endpoint. This intrinsic link with serverless functions makes API endpoints the largest attack surface in which DoW could manifest.

Datta *et al.* [44] proposes a system to mitigate the flow of indirect functions by monitoring the activity of every API trigger of the function. This research was directed at preventing information theft. However, analysis of all exposed API endpoints is necessary for DoW mitigation as any unprotected function triggers will serve as a point of attack.

There are unique vulnerabilities that can target APIs, as outlined in the Open Web Application Security Project's (OWASP) top ten risks to APIs [45]. These vulnerabilities serve as an entry point for an attacker to cause DoW. Specifically in the cases of a *Lack of Rate Limiting* and *Insufficient Logging and Monitoring*, an attacker could flood API endpoints with requests quickly driving up costs. However, all of these vulnerabilities can be exploited with the aim of causing DoW.

Efforts to combat API vulnerabilities are largely based on training developers to identify the issue and implement a fix. To this end, a number of training schemes that gamify vulnerability exploration by rewarding users for correct exploitation of known vulnerabilities on purpose-built insecure applications have been developed [46–48]. OWASP examples of these applications include:

WebGoat[3]: a deliberately insecure application that allows developers to test vulnerabilities commonly found in Java-based applications that use common and popular open source components.

Juice Shop[4]: a sophisticated insecure web application used in security training, awareness demos, and as a test environment for security tools. Juice Shop encompasses vulnerabilities from the entire OWASP Top Ten along with many

[3]https://owasp.org/www-project-webgoat/
[4]https://owasp.org/www-project-juice-shop/

other security flaws found in real-world applications.

2.3.4 Denial of Wallet in Literature

At the time of starting this research in 2020 there was no academic analysis of DoW. A number of blogs have drawn attention to the issue [49–51] although they only go as far as to describe what DoW is rather than conduct in depth analyses of the issue. Pursec [52] released a list of ten security risks that face serverless computing with DoW featured on that list. They do not go into detail on DoW specifically, instead focusing on DoS attacks and suggesting mitigation techniques such as:

1. Write efficient serverless functions that perform discrete targeted tasks.
2. Setting appropriate timeout limits for serverless function execution.
3. Setting appropriate disk usage limits for serverless functions.
4. Applying request throttling on API calls.
5. Enforcing proper access controls to serverless functions.
6. Using APIs, modules, and libraries that are not vulnerable to application layer DoS attacks such as ReDoS and Billion-Laughs-Attack

In 2017 OWASP included DoW in their risks to serverless report [53]. This is one of the first major reports that categorically lists DoW as a threat. AWS also released two whitepapers around this time, discussing the security of its Lambda serverless platform[54] and serverless architecture in general [55], which have seen subsequent revisions. However, no mention of DoW is made.

The available literature on DoW was non-exhaustive, with little to no testing to further understand the potential damage this threat could inflict. In 2021 and 2022 we published two papers on the subject. The former being an initial investigation on DoW in order to give it a formal definition in academia (Chapter 3) and the latter furthering our investigation and polishing our definition along with development of a tool to aid further research (Chapter 5). These publications have sparked what appears to be a growing interest in DoW. The Register published an article summarising our work [56] that garnered good discussion on the article

itself and was widely shared on social media. Two articles published in 2022 by two seperate research groups specifically focus on DoW, one being a continuation of our work [57] and the other investigating a new realm of DoW termed *internal DoW*, where the attack originates within the applications network [58].

2.4 Machine Learning in Cyber Threat Detection

2.4.1 Background

Machine learning is a branch of artificial intelligence that focuses on developing algorithms and models that enable computers to learn from data without being explicitly programmed. Machine learning algorithms can recognise patterns in the data and make predictions or decisions based on these patterns.

Supervised learning is a subcategory of machine learning. It is defined by its use of labelled datasets to train algorithms that classify data or predict outcomes accurately. As input data is fed into the model, it adjusts its weights until the model has been fitted appropriately, which occurs as part of the cross-validation process. Cross-validation is a technique used to assess a machine learning model's performance and generalisation ability. It involves dividing the data into subsets, training the model on some subsets, and evaluating it on others. This process is repeated multiple times, and the average performance is used as an estimate of the model's effectiveness. It helps in avoiding overfitting and provides a more reliable evaluation metric.

Regression finds correlations between dependent and independent variables. Therefore, regression algorithms help predict continuous variables such as house prices, market trends, and weather patterns.

Classification is an algorithm that finds functions that help divide the dataset into classes based on various parameters. A model is trained on a dataset and categorises the data into various categories depending on what it has learned.

Unsupervised learning is a type of machine learning in which the algorithm is trained on an unlabelled data set, that is, the input data is not associated with any output. Unsupervised learning aims to find patterns and relationships in the data,

such as clusters of similar data points or principal components that capture most of the variance in the data.

Reinforcement learning is a type of machine learning in which an agent learns to interact with an environment by performing actions and receiving rewards or penalties based on its actions. The agent learns a policy that maximises cumulative reward over time. Reinforcement learning has been successfully applied to a wide range of applications, such as playing games, robotics, and autonomous driving.

Machine learning is increasingly being used in cyber security to help detect and prevent cyber attacks. With the rise of big data, it has become increasingly difficult for humans to analyse and identify potential threats. Machine learning algorithms can be trained on large datasets of known cyber attacks and patterns of suspicious activity, and then used to detect and respond to new threats in real time. Within cybersecurity, applications of machine learning include [59]:

- **Malware detection** - Detect and classify malware by analysing patterns in code, behaviour, and network traffic.
- **Intrusion detection** - Detect and respond to suspicious activity on networks, such as attempts to exploit vulnerabilities or unauthorised access.
- **Fraud detection** - Detect fraudulent activity, such as credit card fraud or identity theft.
- **Phishing detection** - Detect and block phishing attacks, which use social engineering techniques to trick users into divulging sensitive information.

However, the use of machine learning in threat detection introduces challenges and considerations that, if not addressed, will lead to false or missed threat detections.

- **Data quality** - Machine learning models require high-quality data to produce accurate results. If the data is incomplete, inconsistent, or biased, the model may produce incorrect or unreliable predictions.

- **Adversarial attacks** - Attackers may attempt to manipulate machine learning models by feeding them misleading data or exploiting vulnerabilities in the model itself [60].
- **Explainability** - Machine learning models can be difficult to interpret, which can make it challenging for security analysts to understand how the model makes decisions and identify false positives.
- **Privacy** - Machine learning models may be trained on sensitive data, such as personally identifiable information, which raises concerns about privacy and data protection.

Image Processing and Classification

Image classification is the labelling of images by a pre-trained model. One application of such that has increased advancement in the field exponentially is the *ImageNet Large-Scale Visual Recognition Challenge* (ILSVRC) [61].

The ILSVRC was an annual competition in the field of computer vision that was held between 2010 and 2017. The goal of the challenge is to develop algorithms for object detection and image classification that can accurately classify objects in large-scale datasets.

The challenge is based on the ImageNet dataset, which contains more than 1.2 million images with more than 1,000 object categories, all labelled with bounding boxes and class labels. The ILSVRC is divided into two tracks: detection and classification. The detection track requires algorithms to identify and locate objects within an image, while the classification track requires algorithms to classify entire images into one of the 1,000 object categories.

The challenge has played an important role in advancing the field of computer vision and has led to the development of several breakthrough algorithms. In particular, the 2012 ILSVRC competition saw the introduction of the AlexNet architecture, which achieved a significant improvement in classification accuracy over previous methods.

Neural networks are a type of machine learning process that uses interconnected nodes or neurons in a layered structure that resembles the human brain. It creates an adaptive system that computers use to learn from their mistakes and improve

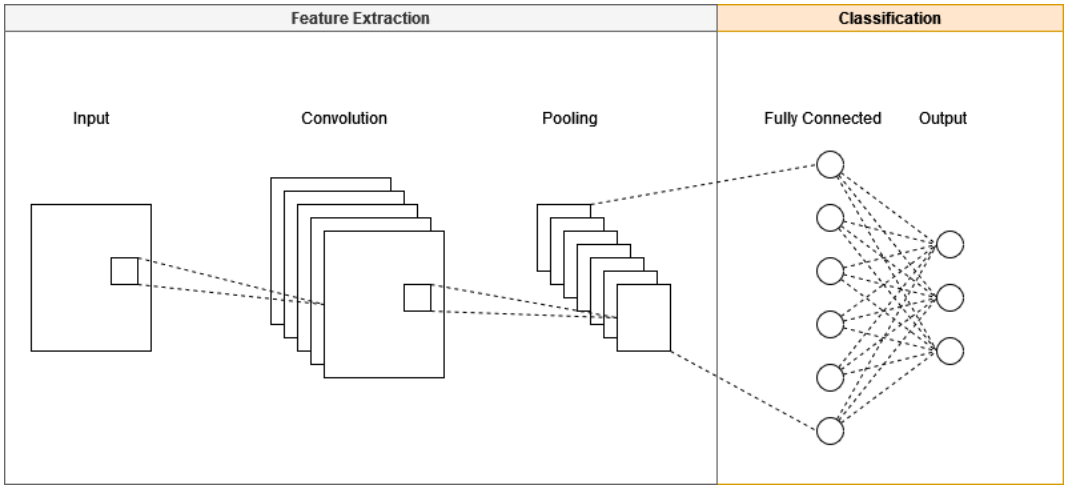

Fig. 2.3 Basic architecture of CNN

continuously. They have consistently been at the top of the image classification section of the ILSVRC [62]. In particular, the use of Convolutional Neural Networks (CNN) has paved the way forward in image classification.

A CNN consists of several layers, including *convolutional* layers, *pooling* layers, and *fully connected* layers. The input to the network is an image, which is passed through a series of convolutional layers that perform feature extraction. Each convolutional layer consists of several filters, which are small matrices that are applied to the input image to extract specific features, such as edges, corners, and textures.

After the feature extraction process, the output of the convolutional layers is passed through pooling layers, which reduce the dimensionality of the feature maps and improve the computational efficiency of the network. The pooled feature maps are then passed through fully connected layers, which perform classification or regression tasks based on the extracted features (Figure 2.3).

CNNs are typically trained using back-propagation, a gradient-based optimisation algorithm that updates the weights and biases of the network based on the error between the predicted output and the true output. The training process involves feeding large amounts of labelled data into the network, and adjusting the weights and biases of the network to minimise the error between the predicted output and the true output [63]. Examples of high performing CNNs in the ILSVRC include:

Visual Geometry Group Simonyan *et al.* [64] of the Visual Geometry Group (VGG) investigate the effect of the convolutional network depth on its accuracy in the large-scale image recognition setting. Their submission to the ILSVRC in 2014 received first and second prize in the localisation and classification sections, respectively. They used small convolution filter sizes of 3×3 in architectures of increasing weight layers, testing from 11 to 19 layers. They found that the networks with 16 layers (13 convolution layers and 3 fully-connected layers) and 19 layers (16 convolution layers and 3 fully-connected layers) achieved 74.4% top-1 classification accuracy on the ImageNet dataset.

Residual Networks He *et al.* [65] developed a model that utilises *residual learning* in order to drastically increase the number of weighted layers from VGG19's 19 layers to 152 layers without the cost of increased complexity. This is achieved by creating *shortcut connections* between stacked layers where the residual representation of the network is generated and then used in the subsequent number of stacked layers. The author's Residual Network with 50 layers (ResNet50) achieved a 75.3% top-1 classification accuracy on the ImageNet dataset.

SqeezeNet Iandola *et al.* [66] sought to replicate AlexNet [67] level classification accuracy with a more lightweight network that is easier to train and deploy. This is achieved via three strategies:

1. Replace 3×3 filters with 1×1 filters
2. Decrease the number of input channels to 3×3 filters
3. Downsample late in the network so that convolution layers have large activation

SqueezeNet achieved 60.4% top-1 classification accuracy on the ImageNet dataset. However, the model was $510\times$ smaller than non-compressed AlexNet.

Xception Chollet *et al.* [68] aimed to build upon the GoogLeNet (later InceptionV3[69]) model by re-interpreting *inception modules*, that are a means of making the process of a convolutional kernel mapping cross-channel and spatial correlations easier by treating those channels as separate spaces. The hypothesis is that cross-channel correlations and spatial correlations are sufficiently decoupled that it is

preferable not to map them jointly. Xception takes this hypothesis to the extreme by basing its architecture on entirely depthwise separable convolution layers. Xception achieved 79% top-1 accuracy on the ImageNet dataset.

MobileNet Sandler *et al.* [70] employ the techniques demonstrated by the previously mentioned models, namely residual representations of the network with shortcut connections and depthwise separable convolution layers. They achieved 74.7% top-1 accuracy with MobileNetV2 on the ImageNet dataset. This background knowledge is relevant as in Chapter 6, these CNNs are trained to detect DoW.

2.4.2 Applications of Machine Learning in Cybersecurity

As cyber attacks become ever more complex and powerful, the need for equally effective mitigation systems increases. To this end, the use of machine learning algorithms trained on datasets of previous attacks for classification of suspicious traffic has become an important step forward in cybersecurity.

Niyaz *et al.* [71] use deep learning for feature reduction of a large set of features derived from network traffic headers for the detection of DDoS. They use a Stacked Auto-Encoder (SAE) which is a type of ANN that consists of multiple layers of autoencoders. An autoencoder is an unsupervised learning algorithm that learns to encode and decode data in an efficient way. The goal of an autoencoder is to learn a compressed representation of the input data, called the latent space, which can be used to reconstruct the original input data. This SAE consists of stacked sparse autoencoders and a softmax classifier for unsupervised feature learning and classification, respectively. A sparse auto-encoder is a neural network with three layers; input and output containing M nodes and a hidden layer containing N nodes. M nodes represent a record with M features. For training, output is the identity function of the input. The sparse auto-encoder networks calculate optimal weights of matrices and bias vectors while trying to learn an approximation of an identity function using back-propagation. Multiple sparse auto-encoders are stacked, so the output of one feeds into the input of the next while also reducing in dimension. Finally, the last hidden layer is fed into the softmax classifier. Normal traffic data was collected by recording use on a home network over 72 hours. The attack data was generated using *hping3* in an isolated environment. *Tcpreplay*

was used on a software defined network environment for training and testing by replaying the recorded normal traffic and generated attack traffic. DDoS attacks were detected with an accuracy of 95.65% using this system.

He *et al.* [72] compare multiple supervised and unsupervised machine learning algorithms to classify outgoing traffic from a cloud platform as DDoS. They train the algorithms to detect the attacks from the source rather than on the victim end. The attacks they chose for detection were: Secure Shell (SSH) Brute Force, DNS Reflection, ICMP Flood and TCP Synchronise Flood.

They trained the following algorithms:

- **Supervised -** Linear Regression, Support vector machines w/ Linear, polynomial and radial basis function kernels, Decision Tree, Naïve Bayes and Random Forest
- **Unsupervised -** K-Means and Gaussian Mixture Model for Expectation Maximisation

They ran these trained algorithms on a test cloud running OpenStack for virtual machine provisioning and found that Random Forest had the best accuracy at 94.96%.

Priya *et al.* [73] also found that Random Forest had the best accuracy, training the algorithm on attacks gernerated by *hping3*.

Ko *et al.* [74] utilised netflow data on an Internet Service Provider (ISP) and BoNeSi [75] for generating attacks. A dynamic feature selector was devised for use with Self-Organising Maps in training an attack detection system.

Examples of cyber threat detection utilising image classification are predominantly on malware detection. Malware can be reduced to greyscale images where a given malware binary is read as a vector of 8 bit unsigned integers and then organised into a 2D array. This can be visualised as a greyscale image in the range [0,255] (0: black, 255: white) [76]. Local binary patterns (LBP) can be extracted from such images in order to classify the malware [77]. This idea is further enhanced by the creation of MalNet-Image [78]. It eases the past difficulty of researching the topic with a large dataset, offering 24× more images and 70×

more classes than existing databases of binary image representations of malware. The role of image classification in cyber threat detection grows as more methods of visualising attack data are devised. Azab *et al.* [79] advances malware classification from binary images and searching for LBPs to representation of malware as spectroscopy images and uses CNNs for classification.

Wang *et al.* [80] create image representations of raw traffic from *pcap* files in order to detect malware traffic. A 4 step procedure is used to trim, sanitise and anonymise the *pcap* file before translating the byte code into an image. They subsequently train a CNN to identify features representing malware and achieved a 99.17% accuracy in their 20 class classifier.

Taheri *et al.* [81] perform a similar process of creating image representaions of raw traffic *pcap* files and CNN training. However, the aim of this research is botnet detection. A classification accuracy of 99.98% was achieved on the CTU-13 dataset [82].

McCullough *et al.* [1] propose a novel method of representing aggregated TCP traffic from the Internet of Things (IoT) as heat maps and subsequently training a CNN to detect DDoS traffic patterns. We take inspiration from this work, adapting the use of heat map representations of traffic for one month's worth of HTTP requests to FaaS endpoint triggers, rather than egress TCP traffic from an IoT devices (Figure 2.4). They were able to mitigate DDoS by detecting if there was a compromised IoT device on the network that performed the attack and removing them. Their detection approach used the same CNN algorithms that were previously discussed above. The authors reported excellent classification accuracy utilising existing models of approximately 99% and F1 scores of approximately 97%.

With the continued persistence of CNNs in all areas of image classification, including cyber threat detection, the use of existing models should be considered, where these models have performed with high accuracy in other domains. The previously discussed selection of notable CNNs produced for the ILSVRC heavily inspired the work presented in our approach to attack detection. Given their success in other domains of computer vision, we concluded that a CNN will serve

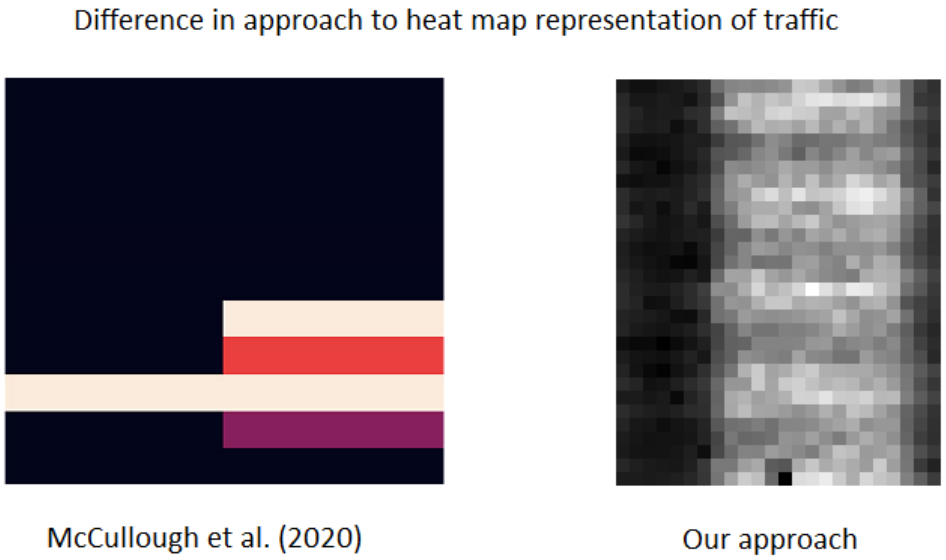

Fig. 2.4 Difference between McCullough *et al.* [1] approach to heat map representation of traffic and ours. The former shows ingress traffic in the left column and egress in the right with each row representing a time step. Our approach maps a month's worth of incoming requests to a 24×30 grid (further discussed in Chapter 6)

as the backbone of our detection strategy. We perform an evaluation of the models listed above in Chapter 6 and further take inspiration from their underlying architecture in the development of our own model, DoWNet.

2.5 Summary

In summary, this background and related work serves as a useful starting point for DoW research. DoW is a variant of EDoS, in which the damage caused is financial as a result of unwanted operations on an application. DoW and click fraud exhibit similar characteristics, such as the ability to cause damage per click/invocation and the fact that it is not necessary to perform flooding-style attacks to have an effect. API endpoints are the largest attack surface for DoW, since REST APIs are the most common trigger for serverless functions. As such, understanding the security concerns of APIs is important. Also, as this work proposes the use

of synthetic data for machine learning classifier training, it is beneficial to look towards existing research on the subject.

Looking at previous uses of machine learning in cyber threat detection, we can begin to form ideas on how best to approach DoW detection. The work done on novel uses of image classification for attack detection inspired the developed solution in Chapter 6.

CHAPTER 3

Denial of Wallet - Defining a Looming Threat to Serverless Computing

This chapter introduces the main focus of this research, the Denial of Wallet attack. This chapter presents a detailed discussion and formal definition of DoW in academic literature. The work outlined in this chapter contains content from the first published definition of DoW in academia **Kelly, D.**, Glavin, F.G., and Barrett, E. (2021) "Denial of wallet: defining a looming threat to serverless computing" Journal of Information Security and Applications [83]. Up until the publication of the article in 2021, all DoW discourse was carried out on technology blogs [49–51] and occasional security reports [52]. DoW was often an afterthought, assumed to be mitigated by standard DoS mitigation. However, as this chapter will discuss, DoW has nuance to the means in which it can operate that would allow for circumvention of typical request rate limiting, thus answering *RQ1*, "What makes DoW unique to other attacks and how does this influence the potential design of its attack patterns?" . As will be discussed in this thesis, DoW on serverless functions as a targeted attack is presently theoretical, as there are no publicly disclosed instances of it. As such, in this research, we have attempted to address the issue of lack of data by proposing basic attack patterns that would be the most obvious starting point in the evolution of DoW. For this, we approached the matter from the point of view of an adversarial actor (black hat). We expand

upon this line of thinking with extended discussion on those proposed attack patterns (addressing RQ4) and potential other cases where DoW may be executed. As a means of balance in our discussion, this chapter also highlights the current mitigation strategies available on commercial platforms.

3.1 Introduction

As previously discussed, serverless computing is an application deployment architecture that aims to provide pay-as-you go event driven functionality. Applications are developed as per each desired process and the event that invokes it. Serverless function platforms provide the infrastructure to deploy code for execution across their cloud and define the event processing logic that prompts the functions to run using the model: *event, trigger, and action.* Serverless computing abstracts back-end management from users, allowing only minimal access to some basic parameters such as function memory allocation and function run time timeout. Functions execute on the platform's traditional IaaS VM offerings. However, the provision of such VMs is managed by the platform in response to function invocation and not by the developer. A function container is spun up on the VM where the serverless function will execute. A container will be made for each function invocation, and containers may be reused for repeat invocations. Unlike IaaS, you do not pay for the uptime and resources consumed by the *execution* VM or the function container, but rather for the run time of each function, hence the name FaaS. This system allows for serverless applications to scale massively as the application is hosted on the serverless platform's cloud and as such has access to its resources. The appeal of decreased time to deployment, lack of servers to manage, and the pay-per-use cost model of the functions that execute the business logic has accelerated its adoption by many application owners. As these serverless function-driven applications are highly scalable, a weak attempt at a flooding-style DoS attack may be absorbed with no disruption to service. However, such a capability also leaves serverless functions vulnerable to an evolution of the EDoS attack. This evolution has been named *Denial-of-Wallet* [83]. DoW can describe any abuse of a pay-per-use cloud product to cause an inflated bill. However, in the scope of this research, we will focus on the continual triggering

of serverless functions in an attempt to induce greater operation costs for the application owner.

The contributions of this chapter are as follows:

1. First formal definition of DoW on serverless computing in academia.
2. Preliminary investigation into potential attack patterns and theoretical damage potential of DoW.
3. Extended discussion on further exploitation of serverless functions for DoW by adversaries.
4. Initial hypothesised mitigation strategies beyond what is presented in this thesis.

3.2 Presenting Denial of Wallet

3.2.1 Formal Definition

The purposeful targeting of an individual's service with the aim of causing undue usage bills has existed under various guises in the past. In this research, we approach the topic of DoW from the perspective of specifically targeting serverless functions. To date, this perspective has received very little attention, and, as such, ground should be broken on this latest avenue for attackers to exploit.

We define DoW attacks as the intentional mass and continual invocation of serverless functions, resulting in financial exhaustion of the victim in the form of inflated usage bills. Execution costs are increased via excessive function run time, invocation count, and additional resource consumption. This can be achieved by exploiting the massive capability for scaling, as the platform will handle the increased load, but will incur costs in doing so or taking advantage of the gradual accumulation of excessive traffic that does not trigger traditional rate limiting systems.

A DoW attack is a unique form of attack that especially applies to serverless computing. Due to the scaling capabilities of serverless platforms, the effects of a regular DoS attack can be dealt with by massively scaling to handle the volume

of requests. However, this scaling comes at a price. Serverless, or Function-as-a-Service, bills the application owner per run time of each function invocation. The resulting volume of function invocations following a DoS attack would therefore incur financial exhaustion of the application owner. As such, the term DoW came to describe this attack, since unlike DoS, where the service is targeted and disrupted, the *wallets* (that is the finances) are the target. The result is an application owner who must now pay for function executions that are not, in fact, *legitimate* usage. As we will discuss in the *Mechanisms of Attack*, simple flooding attacks are not the only threat. We believe that there is potential for a slow rate attack that avoids detection.

The term DoW potentially came into being when it was used on Twitter in 2013 to describe the excessive bills posted by a user for leaving a BizTalk VM running [84]. The term has since become popular to describe any instance of runaway usage bills following some unexpected circumstance (e.g. infinite loops or VMs being left active when not in use). DoW as a targeted attack has been on the conscience of developers even before the advent of FaaS.

One of the earliest uses of the phrase in a publication is in the OWASP report on the security risks of serverless computing [53]. It is mentioned as a risk to consider along with traditional DoS attacks. Prior to this, DoW was often referred to as *EDoS*, a *financial exhaustion attack*, *denial of capital* or simply *serverless denial of service*. However, we believe that a *serverless DoS* attack and a DoW attack can be seen as unique threats, with the former focusing on serverless specific resource exhaustion and the latter being as we have described.

3.2.2 Preemptive Examination of a Threat

Initially, there was little to no targeted research on DoW; however, the area has started to see increased interest with the publishing of papers in late 2022 [58, 57]. We believe this is due to the absence of publicly known occurrences of attacks. However, smaller scale exploits have been reported in the media of the supposed successful takedown of a Dallas Police Department *antisocial behaviour reporting* application by flooding the AWS storage bucket with Kpop videos [85], resulting in excessive usage bills [86]. This presents a unique opportunity to be ahead of attackers by mitigating against attack methods that are early in development. The vulnerability exists in serverless technology, and it would be naïve to believe

Table 3.1 Comparison of related attack terms

Attack Name	Description
DoW	Intentional abuse of serverless functions to cause inflated usage bills
Financial exhaustion attack	*Catch-all* term that can describe any attack that causes inflated usage bills
Denial of capital	Lesser used term the same as financial exhaustion attack
EDoS	Flooding attacks on cloud services, such as VM provision, that prompt massive scaling causing inflated bills
Serverless DoS	Attacks targeting serverless function seeking to disrupt service and not specifically cause financial damage

that it will not be exploited. In Section 3.4 we discuss the potential motivations for conducting such an attack and the potential scenarios in which they may be executed.

Research on DoW will encompass strategically approaching the issue from the perspective of a bad actor. This domain shares many key attributes with bot detection, DoS mitigation, and cloud security, meaning it also inherits their pitfalls and difficulties.

This research aims to bring a dormant threat into the conscience of the community and potentially address it before it may even begin and develop effective mitigation strategies before it can become a prominent threat.

3.3 Mechanisms of Attack

In the previous chapter, we discussed related attacks such as DoS, EDoS and Click Fraud. Since there have been no reported major instances of DoW, it is necessary to consider attack strategies that may be developed. In this section we will take precedent from the techniques employed by DoS attacks and analyse their potential for modification as a DoW attack. Many of the strategies used in DoS attacks are viable in a DoW setting. However, we propose a number of unique cases that invariably distinguish DoW as its own potential threat.

3.3.1 Traditional Attack Methods

As serverless computing operates on the *No Ops* principle (that is, the developer is not involved with the setup or maintenance of the application's back-end), popular traditional DoS attack patterns such as ICMP Flood, SYN Flood and UDP Flood [33] may not necessarily work as functions themselves do not possess the infrastructure such attack's target. If the goal is to specifically target serverless functions rather than taking down an application as a whole, the most appropriate attack method would be HTTP flooding given that the majority of serverless functions work from API triggers. We distinguish between short-time span and long-time span HTTP flooding below.

Short Time Span Attack (Flood)

Short time span attacks are any form of *flood* attack. As many requests as possible are sent to the server as quickly as possible. This is the basis of DoS. A Web Application Firewall (WAF) is the recommended mitigation approach for large spike HTTP flooding attacks. An effective strategy against HTTP flooding DoS attacks is rate-limiting requests from a single IP address via the WAF. Commercial platforms offer WAF services that implement safety against OWASP's top 10 security risks [53] and provide rate limiting rules such as AWS[87], Google Cloud[88], and Microsoft Azure [89]. These rules are difficult to establish. A hard limit will protect against short time span (large spike in requests) HTTP flooding, but at the risk of producing false positives in the event of actual real traffic spikes. Adaptive rules could be deployed but would become a target for long-term false normative baseline inducing attacks.

Rate limiting also exists within the API Gateway services themselves. These are generally limits that show off the service's ability to handle large traffic (AWS API Gateway can handle 10000 requests per second[90]). Unlike the rate-limiting rules on the firewall, if the API rate limit is reached, this could incur a DoS result as resources are taken up. Efforts to stop the effect of DoW by limiting the massive scaling factor of serverless applications could instead cause DoS as limits are reached.

Long Time Span Attack (Leech)

There are a number of *slow* DoS attacks, as described by Zargar *et al.*[91], such as Slowloris and R-U-Dead-Yet (RUDY). These attacks, through various means, aim to send high-workload requests that lock up connection resources i.e. sockets [92].

A new strategy may be employed for long-term DoW attacks that differs from the previous interpretation of *slow* attacks. We theorise the idea of *leeching* as a potential form of DoW. A leech being a malicious programme that continually triggers API endpoints that invoke functions. These leeches operate indefinitely and at a rate that would not be detected by reasonable WAF rate-limiting rules. The key difference between DoW and DoS is that DoW does not need to run all at once to consume all available resources to cause damage. As such, a leech DoW attack could be executed over a long time span and appear as legitimate traffic. The leeches will then appear as regular users. If there are enough leeches, then the damage increases. Unless you have some reason to believe this is illegitimate traffic, then the traffic will be allowed to continue and the bill will be paid.

The pay-as-you-go pricing model that commercial serverless platforms operate is the vulnerability that allows for such an attack. If the load of a DoS attack was distributed over a month, the attack would be ineffective as the load on the system would never become high enough to bring it down. However, with FaaS, the bill continually rises. This attack plays on the same vulnerabilities as *click fraud* does in that it is up to the accounts manager to spot the anomaly in the billing, which as shown through the prevalence of click fraud, occurs on a large scale.

In DoS mitigation systems that use an adaptive rule set, i.e., can change the firewall rules based on fluctuation in *deemed to be real* traffic [37], a DoW attack can again utilise the fact that it can be executed over a long time span. Unlike a rigid firewall rule where it will be forever limited, if the attacker gently increases the intensity in such a way that the mitigation system adapts to it, it will then establish new false normative baselines.

3.3.2 Distributed Denial of Wallet

A long time span attack would be the most simple method of bypassing WAF rules and implementing a DoW attack. However, a single attacker or *leech* may not inflict financial damage on a scale that would affect larger organisations. Multiple leeches could be deployed onto a botnet either willingly, such as through IRC chat as seen in Operation Payback [93], or via malware that makes unknowing victims hosts for the leeches. A distributed approach to DoW would ultimately prove difficult to detect given how closely it would resemble real traffic using a long time span attack protocol.

3.3.3 Serverless Exploitation

As well as API triggers, serverless functions can also be invoked by uploading to a storage service (such as S3 buckets for AWS Lambda). This introduces the concept of serverless specific attack patterns such as function input parameter exploitation, by flooding a storage service with images of varying size depending on the limits imposed by the function.

To fully realise the potential harm a DoW attack may be capable of, it should not be simply seen as a slow-burning DoS attack. Instead, it should fully exploit any oversights the developer may have made. If no limit is imposed on functions that process files, the run time of a function will increase. We show this effect in Section 3.5.1.

Similarly, on Microsoft Azure Functions, the developer does not configure the memory allocation of a function unlike its competitors. This could similarly be abused by forcing a function to scale to a higher memory allocation to cope with larger inputs.

3.3.4 Fake Users

A novel approach that could be taken towards performing a DoW attack would be the mass generation of fake users on a user subscription web application. Varol *et al.* [94] found that between 9% and 15% of Twitter accounts are bot accounts. This equates to roughly 49.5 million accounts based on the current monthly active

user statistics in 2020 [95]. Using a tool like Selenium[1], which automates browser activity, a mass of clients could perform a site walkthrough creating many fake users. Not only would this trigger many functions involved with profile creation but also, as discussed in Section 3.3.3, uploading the largest images possible, e.g. as profile pictures, can be used to increase run time. A further result of the mass generation of fake users would be access to other API endpoints such as page querying or other site functionality, further falsely invoking more functions causing more damage.

In recent studies, there have been three main approaches to fake user detection: graph-based, crowd-sourcing, and machine learning.

Graph-based detection is the use of graphs constructed from the social network of users to understand the network information and the relationships between edges or links across accounts to detect bot activity. There are many approaches to graph-based bot detection, a popular one being the use of *random walks*, for example SybilRank [96], Criminal account Inference Algorithm (CIA) [97] and SybilWalk [98]. SybilWalk claims a more robust classification with 99% of the top 80,000 nodes ranked in order of likelihood to be a Sybil (bot), being correctly classified Sybils. However, in the ranking lists produced by SybilRank and CIA, only 0.3% and 30% are Sybils, respectively. Accounts are ranked by performing a random walk-based movement on an undirected social graph. The idea of the random walk method is to label human users with benignness scores and Sybil users with badness scores. The score is then used to rank users.

Crowd-sourcing is the use of humans to label whether a given user is a Sybil or a human. They identify, evaluate, and determine behaviours that would point towards an account being a bot. Alarifi *et al.* [99] recruited 10 volunteers deemed to have expert knowledge of Twitter (all BSc Computer Science graduates and active Twitter users) to rate and label 2000 random accounts captured via a Tweet streaming API. Using this, a ground truth data set was established for the further training and comparison of various machine learning algorithms such as C4.5 and Random Forest.

The machine learning method involves developing algorithms and statistical methods that can develop an understanding of the revealing features or behaviour of social network accounts in order to distinguish between human and computer-

[1]https://www.selenium.dev/

led activity. Ersahin *et al.* [100] used a Naïve Bayes based classification algorithm on a data set that had undergone preprocessing via Entropy Minimisation Discretization (EMD). The results were 86.1% correct classification before EMD and 90.9% after.

3.4 Motivation for Denial of Wallet

Attacks such as DoW and previous examples of EDoS differ from destructive attacks such as DoS in that the resulting damage has a large range of values in which it can be effective. A successful DoS attack will shut down a service and be immediately noticed by victims. However, with DoW, the financial damages may accrue over a long period of time depending on how perceptive the application owner is. To this end, DoW may affect smaller companies and individual application developers more due to limited funding. The ability to potentially remain undetected for longer is also a strong advantage to using such an attack.

In a recent report by Trend Micro [101], the price of botnet acquisition has decreased dramatically since 2015. Generic DDoS botnets can be acquired from $50 a day and generic botnets can be rented from $5 a day. Therefore, it is highly feasible that, despite the initial investment, an attack could net more damage.

DoW is an attack that may be chosen for its covertness and suitability for sabotage. Rather than an attack that seeks to *take out* a victim, they can potentially weaken them. This could be a tactic employed where a larger entity is interested in stifling the growth of a smaller one. It may also be used to cause customers to distrust the application if it is regularly attacked.

Another possible motivation would be as a form of ransom attack. Attackers may cause a DoW and then contact the victim looking to extort some payout with threat of continuing to cause DoW or even contact in advance with threats of a DoW in order to receive a payout. We discuss this in more detail in Section 3.6.

3.4.1 Scenarios

For DoW to occur, a malicious entity must gain access to function endpoints. This can be broadly achieved via knowledge that certain services use serverless functions, e.g., Seattle times image loading [102]. More targeted approaches could

be taken to extract specific endpoints of functions via web scrapers looking for direct API links that follow the typical naming convention, e.g.

- **AWS** - *https://xxxxx.execute-api.xx-xxxx-x.amazonaws.com/xxxxx/xxxxx*
- **Azure** - *https://xxxxx.azurewebsites.net/api/xxxxx*
- **GCF** - *https://xx-xxxxx-spherical-plane-xxxxx.cloudfunctions.net/xxxxx*

It is good practise among industry professionals not to leave API URLs visible like this; however, such a vulnerability could still exist through human error.

The core mechanism of a DoW attack will be to spam function triggers. We consider two likely scenarios for this to occur.

Scenario 1 Function API endpoint URLs have been scraped or leaked. These URLs are loaded into spamming software that continuously hits these endpoints with HTTP requests. The rate of attack can vary depending on the security of the application. In the presence of no DDoS security (rate limiting, etc.), fast paced attacks can take place but will raise suspicion when an application owner sees a sudden spike. Slow-rate attacks, *Leeches*, are effective for DoW due to their difficulty of detection. They are akin to *Bad Bot Traffic*, fictitious traffic generated by botnets intended to give false analytics. As such, slow rate DoW has a double impact of driving up operational costs with function invocations and spoofing application owners with falsified traffic (potentially causing further financial damage when used to justify increased spending).

Scenario 2 Developers have successfully hidden or obscured API URL endpoints or the endpoint is a different trigger such as upload to a storage system. Attacks will need to be performed as a fake application user. Web application testing software such as Selenium can automate interactions with a web application. Applying a similar tactic to a botnet will produce fictitious traffic as with the URL example. The downside to this is that it requires greater computing resources and knowledge of which applications use serverless functions and how they are triggered.

Table 3.2 Effect on Run Time of increasing Image Size

Image size $l \times w$ (pixels)	Test 1 (ms)	Test 2 (ms)	Test 3 (ms)	Test 4 (ms)	Test 5 (ms)	Average (ms)
540	248.27	230.51	254.77	239.32	236.22	241.818
1080	371.18	368.43	361.15	385.22	409.24	379.044
2160	1045.59	1051.62	1083.57	1046.05	1059.11	1057.188
4320	2945.17	2958.46	2954.22	2963.07	2933.42	2950.868
8640	9901.85	9843.55	9924.92	9877.68	9881.59	9885.918

3.5 Theoretical Damage Analysis

In order to convey the threat of DoW we will theorise a worst case example of the attack based on a plausible situation. This damage analysis is to demonstrate that DoW has the capacity for large scale financial impact.

3.5.1 Serverless Function Input Parameter Exploitation

A common use case of serverless functions is in image re-processing, such as creating a thumbnail from a user uploaded image [103]. We use AWS Lambda as an example to demonstrate how one could exploit input parameters to a serverless function by uploading large images that must be resized to a thumbnail size. For this, an AWS Lambda function was created that resizes images uploaded to an AWS S3 bucket to a 128×128 pixels thumbnail [104]. The thumbnail image is then stored in its own separate bucket. Images of increasing size were uploaded, and we observe the effect on function run time. The function memory allocation was kept constant at 512MB, as we found that lower memory allocations could not handle the largest images. The results are shown in Table 4.1.

As expected, there is an increase in run time with an increase of image size. In this scenario, it would be up to the developer to impose a hard limit on image size. However, as smartphones are producing higher resolution images with each new flagship device, for example, the baseline of an acceptable maximum image size gets pushed higher. This makes even well-thought-out functions susceptible to input exploitation.

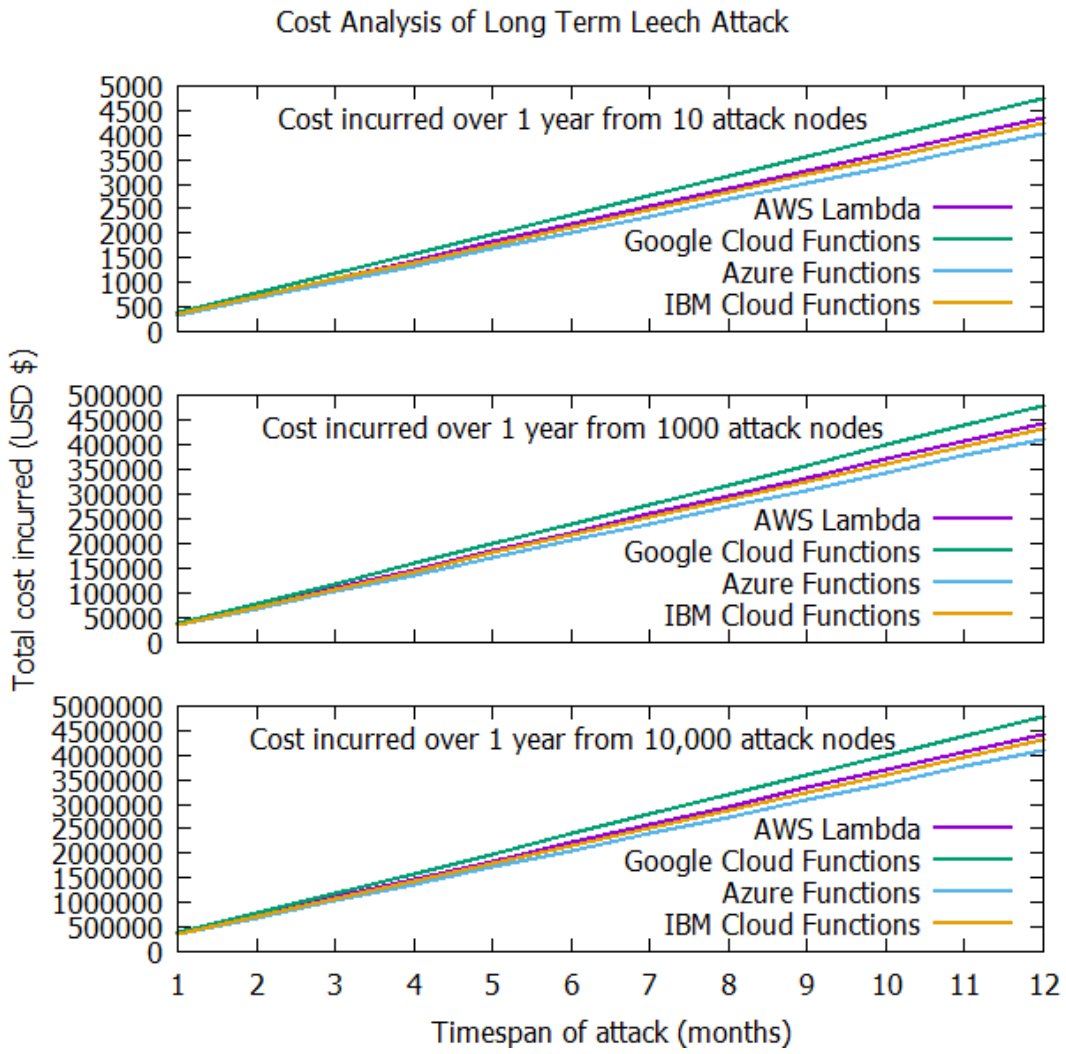

Fig. 3.1 Cost incurred when an increasing number of nodes send 2000 requests every hour

3.5.2 Cost Analysis of excessive Function Invocation

As a reference, an Apple iPhone X takes photos that produce an image of 4032 × 3024 pixels [105]. Cross-referencing our previous experiments on input exploitation (Section 3.5.1), we can take a run time of 2950 ms of a memory allocation of 512MB as a benchmark for a legitimate function that a malicious user would consider targeting for a DoW attack.

A cost analysis of this function on the four largest commercial serverless platforms; AWS Lambda, Google Cloud Functions, Microsoft Azure Functions and IBM

Cloud Functions, was performed for increasing number of function invocations using the serverless function cost calculator we devised as the mock application on our isolated testbed (Chapter 5).

This calculator takes the number of function requests, the run time of each of those functions, and the memory allocation of the function as input and returns the total cost of executing those functions inclusive of all free allowances granted by each platform and additional costs such as API call costs. The official cost guides for each platform were used; AWS Lambda[2], Google Cloud Functions[3], Microsoft Azure Function[4] and IBM Cloud Functions [5].

The three graphs show a linear increase of cost incurred with Google Cloud functions, resulting in the greatest charges followed by AWS Lambda, IBM Cloud Functions, and lastly Azure Functions. With a modest botnet of 1000 nodes, a slow rate attack of 2000 requests per hour will cost an application owner roughly $40,000 after one month and between $400,000 and $500,000 if left unchecked for a year. A botnet of 10,000 nodes will do the same damage in one month that 1000 nodes would do in a year.

3.6 Black Hat Perspective on DoW Execution

A black hat is an individual or group who wishes to inflict cyber attacks for monetary gain, ideology or simply for fun. We, as the white hats, are in constant competition with them for finding exploits and vulnerabilities to attack or defend. For either side to launch successful campaigns, the opposing perspective must be taken in order to theorise counter strategies. This section expands on the discussion of potential DoW attack pattern above for consideration in future research. We make reference to DoWNet, which our DoW detection system proposed in Chapter 6, in order to give context to the points raised.

[2]https://aws.amazon.com/lambda/pricing/
[3]https://cloud.google.com/functions/pricing/
[4]https://azure.microsoft.com/en-us/pricing/details/functions/
[5]https://cloud.ibm.com/functions/learn/pricing

3.6.1 Beyond Hypothesised Attack Patterns

The attack patterns discussed in this thesis serve as a base for formulating various tools and approaches to DoW protection. However, it is unlikely that an adversary will launch an attack as simple as this. These attacks were proposed *leech* attacks, persistent attacks that do not trigger rate limits that may establish new false normative baselines for expected traffic. We believe that *leeches* are one of the greatest threats from the point of view of DoW targeting serverless computing.

For a *leech* to be most effective, it must appear as normal traffic so as not to arouse suspicion. We found during our testing of DoWNet that our simple attack patterns were often detected as they continued their rate of request spamming into the lull hours of the expected traffic volumes. Therefore, for an effective *leech*, an adversary must perform some level of recognition of the traffic patterns of the victims. This will enable them to increase and decrease their attack rate with lulls and peaks, avoiding detection for as long as necessary.

While we have determined that traditional flooding attacks may not be as effective, there may be a case to be made for intermittent high-rate flooding. As discussed, an effective strategy for DoW is to convince the victim that they are experiencing legitimate traffic. A constant barrage of requests lasting many hours will not only be detected by DoWNet, but will also look like a run-of-the-mill DoS attack, therefore prompting the victim to take action. If short bursts of high load traffic are sporadically launched at strategic times throughout a month, it would be detected by DoWNet as obvious spikes in traffic. Multiple short bursts of traffic may convince the victim of legitimate peaks in traffic rather than a DoW attack.

In short, there are numerous ways our initial attack patterns can be expanded upon by a black hat actor to cause more damage and avoid detection. Our examples assume a certain level of protection in place, such as a WAF or even a deployment of DoWNet. However, as serverless becomes increasingly adopted as an easy means of deploying functionality to a web facing service, there will be an abundance of low hanging fruit in the form of unprotected endpoints that, if found, are open to abuse by regular HTTP flooding attacks. This is especially relevant to inexperienced programmers pushing code to code repositories like GitHub. It has been shown that it is a simple matter of scraping public repositories for access credentials that may have been pushed mistakenly [106]. It stands to reason that these same scrapers could also include serverless endpoint APIs

in their search. These inexperienced programmers may also be thoroughly ill equipped to deal with an attack and, as such, may have to terminate their service and/or pay an extortionate bill.

3.6.2 Expanded Attack Surfaces

Serverless functions are most commonly triggered via API endpoints. This has been the focus of our research since it is the most widely used method. However, as mentioned in Section 3.2, there are other means of exploiting serverless functions in order to carry out a DoW attack. That is, abuse of a storage service that triggers a function to action on the uploaded file. We demonstrated this with an image thumbnail generator by uploading more and more large images.

It is also possible to trigger functions that continuously poll a database. Therefore, a similar approach to storage service abuse can be taken, though rather than seeking to increase runtime, the aim is increased invocations. This may also have a knock-on effect if the database charges per access. This brings up the expanded domain of DoW, that is, on other pay-per-invocation services. Cloud providers are increasingly offering this billing model on their services. This may include, but is not limited to:

- **Database service** - Charged per read/write
- **Internet of Things service** - Charged per message
- **Machine Learning service** - Charged per prediction

Therefore, a black hat wanting to launch a successful DoW attack will target as many of these services as possible in order to maximise damage.

3.6.3 Ransom Style Attacks

All attacks mentioned in this work have focused on the immediate damage caused by covertly (in the case of *leech* attacks) or openly (DoS style flooding attacks) spamming function triggers. However, as with DoS, the effect of DoW could be extended by using it as a means of ransom attack. In DoS ransom attacks, the attacker hits a service causing an outage and then follows up with a ransom threat to the victim that they will continue to take down their services unless some

amount is paid. DoW ransom may work in the same way where the attacker threatens to continuously spam function endpoints until a ransom is paid. While DoW does not have the immediate scare factor of taking down a service, it does, however, directly target a service's profit. Also, as of now, the abilities of DoW are unknown; therefore, if an attacker threatens that they can ramp up the attack, the victim has no precedent to fall back on in establishing whether or not this is possible. The victim then has a number of choices:

1. Pay the ransom. The thinking being that the ransom is worth less than the potential damage that may be inflicted.

2. Take down the affected endpoints and functions. This stops the attack, but inadvertently causes DoS as the service has now been disrupted.

3. Change the endpoints. This will stop the attack. However, if the attacker could find the endpoints before, perhaps they can find them again.

4. Do not pay the ransom and absorb the damage cost. This will cause unknown damage but potentially give time to figure out other remedies for the problem.

As a ransom attack, it may not be as intimidating as DoS. However, as it does not require the computing power of DoS, it is cheaper and easier to execute. This may lead to its adoption by *script kiddies*[6] who are looking to cause trouble and potentially make quick money as *hackers-for-hire* on the Darknet.

3.7 Mitigation Strategies with Current Serverless Infrastructure

Securing a serverless function from threats, not limited to DoW, falls in the hands of the application developer. AWS highlight a *shared responsibility model*, where the developer is responsible for the security of the code and explicitly states that poorly configured code resulting in adverse repercussions may not receive sympathy when reported [55]. Steps to protect applications are highlighted in their

[6] Entry level hackers who predominantly use existing tools to perform attack as their knowledge is only basic. Often they are teenagers performing the attacks for fun rather than a higher purpose.

whitepapers to develop well-architected frameworks and the security overview of AWS Lambda [54, 55]. The relevant *pillars of a well architected framework* that should be implemented to protect a serverless application from DoW (specifically for AWS based applications) include:

1. *Operational Excellence Pillar* – Developers should set individual alarms for separate Lambda functions so that you can catch invocations running for longer than usual. AWS X-ray tracing should also be utilised for monitoring of interactions with the service.

2. *Security Pillar* – Applications should undergo risk assessment and have appropriate mitigation strategies. Identity and access management is an important factor in the implementation of secure applications. As serverless functions are often invoked via API triggers, the following mechanisms are recommended:

    - *AWS IAM authorisation* – Only allow users within AWS environments access with IAM credentials (protect against internal attack by a compromised user). Always give the least amount of access.
    - *Cognito user Pools* – Built in user management or integration with social login. Uses Oauth scopes.
    - *API Gateway Lambda authoriser* – Use a lambda function to interact with some Identity Provider.
    - *Resource policies* – Resource policy can block/allow specific AWS accounts, IP ranges, Classless Inter-Domain Routing (CIDR) Blocks, virtual private clouds, etc. via JavaScript Object Notation (JSON) policy file.

3. *Cost Optimisation Pillar* – Application owners should appropriately monitor costs to further refine return on investment. With many more functions and API endpoints, it is recommended to allocate costs from bills to individual resources to gain a more granular view. This allows for quick identification of services that generate elevated costs.

While these safeguards are described specifically to AWS's offerings, the same principles apply to all platforms.

On the development side, the following considerations should be taken at certain levels:

- *Identity and Access* - Implement some form of authentication to hide function interaction endpoints from the public facing Web.
- *Code* - Use of secrets to maintain the privacy of credentials and business critical data. Input validation to minimise risks associated with API requests. Vigilance in monitoring and dependency vulnerabilities.
- *Data* - Encryption in Transit, such as Secure Sockets Layer (SSL).
- *Infrastructure* - Configuration of usage plans to help against effects of DoW. Implementation of a WAF for protection against other forms of cyber attack.
- *Logging and Monitoring* - Appropriate usage of both to detect suspicious usage on the application.

The listed safeguards serve as a possible first defence against the possibility of being affected by a DoW attack. However, these safeguards are not apparent for *new-to-serverless* developers. Official sample applications that such a developer would follow when learning how to create serverless applications [107, 108] are susceptible to API vulnerability attacks, such as capturing an endpoint URL (Figure 3.2) and replaying the request with varying header and body fields to force repeat invocations. Serverless DoW relies on repeatedly triggering functions, the most common method being via API endpoints. Requests can be spammed using trivial programmes that quickly loop REST commands. An attacker is not required to use a large number of nodes to cause damage, as in DDoS. The attack can cause financial damage with only a modest-sized attack source. However, it is worth noting that in the event of such an attack, if the application owner becomes aware and decides to shut down an endpoint or even the whole application, this is essentially a successful DoS with a fraction of the required firepower.

Further to the best practices discussed, there are a number of *Cloud Security Services* that provide DDoS protection. These are either offering of the cloud platform itself, such as AWS Shield[7] or Google Cloud Armor[8], or external services that

[7]https://aws.amazon.com/shield/
[8]https://cloud.google.com/armor

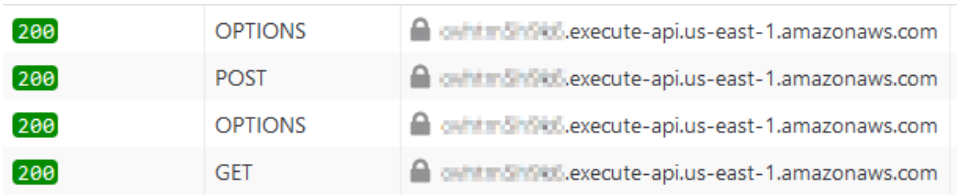

Fig. 3.2 API endpoints that trigger functions can be sniffed on the network by searching for URLs that match the platform template (unique ID blurred for privacy)

operate as a middleman to analyse traffic on an application, such as Cloud Flare[9] or Akami[10]. While DDoS protection would do little to prevent leech attacks given their vastly different attack rate, these services are an effective safety measure by entrusting the handling of an application's security to a tailor made tool and/or expert third party. These DDoS protection services are also prime candidates for deploying DoW recognition algorithms as they are already monitoring all the traffic to an application. We believe that such an offering, whether it is built on the findings of our research or not, will soon become mainstream.

3.7.1 Potential Future Mitigation Strategies

The successful classification of malicious traffic patterns by DoWNet is a great step forward for DoW mitigation research. Our results suggest that the answer to RQ3 "Can an attack that so closely mimics regular traffic be detected and mitigated against?", is in fact yes. Following on from our use of image classification in DoW detection, we have theorised two other potential avenues that further detection research could take.

Function Invocation Graphing In order to conduct anomaly detection, there must be some knowledge of what is *normal*. In serverless applications, there can be a wide range of serverless functions executing the business logic. These may be functions that are triggered one after another in a chain starting from one trigger or there may be many stand-alone functions that could be separately invoked. In the development of DoWTS in Chapter 5, we explain the former, i.e. chains of functions that operate together to fulfil a single user interaction. These functions

[9]https://www.cloudflare.com/en-gb/ddos/
[10]https://www.akamai.com/solutions/security/ddos-protection

are always executed in series. However, in the latter scenario where there are many separate functions, we propose a potential means of detecting anomalous behaviour by creating *normal function invocation graphs*. These graphs would map out the typical interaction with the functions. Therefore, it would be possible to detect when a user is behaving uncharacteristically.

For example, an application that resizes an image may have four public facing functions; *login*, *logout*, *upload* and *download*. A *normal function invocation graph* that may represent typical behaviour on the application may map: *login* -> *upload* -> *download*. If an attacker deploys bots to repeatedly spam whatever function triggers it can find to increase the number of invocations, it may unceremoniously requests functions like: *login* -> *logout* -> *login* -> *logout* or *login* -> *upload* -> *upload* -> *upload* repeatedly. These kinds of interactions should then be easily detectable and flagged for the user to be potentially banned.

User Network Graphing Building on the idea of function invocation graphing and work carried out on social media bot detection via graph-based detection [96–98], the two could be combined to further detect which users may be bots spamming function endpoints. Where anomalous behaviour is detected, the corresponding function invocation graph of that behaviour can be searched for in other user interactions. Where there is overlap in those atypical behaviour patterns, a network of users can be created where further information can be utilised in order to most effectively secure the application. By discovering as many of the suspected attacking bots as possible, it would then be possible to alter security policies based on the shared information of this group, such as region, user-agent, address range, etc.

3.8 Summary

The intentional mass, and continual, invocation of serverless functions, resulting in financial exhaustion of the victim in the form of inflated usage bills, is the formal definition of DoW first published in academia that has seen citation by current researchers taking on the topic [57, 58]. We answer *RQ1* "What makes DoW unique to other attacks and how does this influence the potential design of its attack patterns?" through the following observations. While DoW can be achieved via simple HTTP flooding, similar to DoS attack, we have theorised a

slow rate attack that could be utilised as a covert means of undermining competition or generally causing damage without being detected. There are similarities with better documented attacks, such as click fraud, EDoS, fake users, and other distributed attacks. Understanding these attack vectors can help to conduct a better pre-emptive investigation on DoW. The theoretical damage analysis we performed highlights the financial damage that could occur as a result of DoW. Our simple attacks incurred theoretical damage of more than one million dollars. While these numbers seem outlandish, it is an important baseline for worst-case scenarios that can be built upon to prevent such attacks becoming a reality.

Additionally, we address *RQ4* "What is the potential direction DoW may take in the hands of malicious actors for increased damage and success of attack?" by opening discussion on how a black hat actor may think in order to formulate the most effective DoW strategy. We theorised attack vectors that go beyond the scope of the three that will be tested in this thesis (Chapter 5 and 6). Such vectors would play to the ability to fool an application owner into thinking they are receiving legitimate increases in traffic. We also discussed how DoW is a threat beyond the subject domain of this thesis, serverless computing. Any pay per invocation service is vulnerable to such cost inflation attacks. Finally, we suggest the potential for DoW to be used in a ransom attack situation. Given the relative recency in the uptake of serverless computing and a lack of precedent to fall back on, application owners may feel it to be wise to pay a ransom if being hit with DoW for fear of even greater damage being caused.

In addition to the black hat perspective, we also provide the defence perspective. We outline existing methods of securing serverless functions and also theorise some potential future methods that utilise graphing the usage characteristics of an application and the users interactions on the application.

The following chapters of this thesis will detail our approach to DoW detection.

CHAPTER 4

Behind the Scenes of major Serverless Platforms

All successful cyber attacks require knowledge of the target system. Therefore, it is important to have a complete picture of the infrastructure and possible exploits that could leave a system vulnerable. As serverless computing operates on a *black box* model where end users are not expected to interact with or have knowledge of this infrastructure, we believe it is important to document the current state of the commercial offerings. We provide an in-depth look at the underlying infrastructure of a selection of the largest commercial serverless platforms and how that investigation led us to uncover the danger of runaway usage bills as published in **Kelly, D.**, Glavin, F.G., and Barrett, E. (2020) "Serverless Computing: Behind the Scenes of Major Platforms" IEEE CLOUD Conference [109].

4.1 Introduction

Our investigation on DoW begins with a look behind the scenes of serverless functions and offers up-to-date insight on the hardware differences of the four largest platforms: AWS Lambda, Google Cloud Functions, Microsoft Azure Functions, and IBM Cloud Functions, as well as outline performance metrics and potential effects on performance. Through this analysis, we came to experience the potential threat of DoW. This investigation also serves to highlight the still relative ambiguity that

serverless functions operate under on commercial platforms, where developers must trust in the *black box* that their business logic is executed.

This analysis serves as available counter-intelligence against attackers who may leverage this same information in order to exploit serverless functions to cause excessive usage times. As the commercial platforms do not publish this information themselves, our findings may help inform developers in choosing a serverless platform or in making any other security decisions.

We design a testing system to expose the underlying infrastructure on which serverless functions run. We gather the CPU specifications and usage statistics in order to provide this overview of the resources provisioned. To this end, we establish a means of identifying VMs to create meaningful profiles of VM configurations. From this, we gain insight into their topology and observe the major differences between the platforms. Users cannot directly request the specification of these execution VMs, only the memory allocation each function has access to and, as such, document how this configuration can effect the specification of the VM deployed. We further investigate how this memory allocation affects function performance by recording various timing statistics and benchmarking metrics associated with function execution such as: function runtime, *cold start* time (initialisation lag), CPU utilisation, and disk I/O throughput. Lastly, given that all the functions for a given platform are executed from the same cloud space (split into regions), we investigate the effect of potential interference from load on the platform by running our test system over the course of one month in order to detect any anomalies in runtime performance.

There are countless blogs written by serverless application developers that give a brief overview of the performance of serverless functions [110–114]. The focus of these is often on the issue of *cold starts* and the choice of programming language for the development of functions. Given how quickly these articles are produced, the results often seem at odds with each other as time goes on. For example, Voijta [114] concluded that the memory allocation of a function does not affect the cold start time, while Cui [111] concluded that it did one year later. In academia, a number of researchers have attempted to assess the performance of various serverless platforms. Hendrickson *et al.* [115] created an open-source serverless platform using the *Lambda Programming Model* (develop functions that respond to events), which achieved lower latency at low loads and was better

at bursts of traffic compared to AWS Elastic Beanstalk[1]. The authors also proposed a benchmarking tool (LambdaBench) for the Lambda Programming Model. McGrath *et al.* [116] analysed the performance of AWS Lambda, Google Cloud Functions, Microsoft Azure, and Apache OpenWhisk against their own prototype serverless platform. They made observations on the scaling and cold start latency of the platforms, finding AWS and Google to be best-in-class at the time. However, the testing setup may not have been optimal for recording cold start times i.e. not allowing for enough time to pass where a cold start would be encountered. Llyod *et al.* [117] expand on the cold start issue by defining them as: *provider cold* (first request to the cloud provider), *VM cold* (first request to a VM within that cloud), and *container cold* (first request to a container within that VM), as well as voicing the need for a standard means of benchmarking serverless functions inspired by the suggestions of Aderaldo *et al.* [118] on microservices. Mohanty *et al.* [119] investigate the popular performance metrics used by other researchers with their study focusing on open source frameworks such as OpenFaas, Kubeless, and Fission. There were no comparisons to the commercial offerings in this work. Hellerstein *et al.* [120] provide a *devil's advocate* opinion on serverless architecture, listing limitations such as limited function lifetime, I/O bottlenecks, no specialised hardware, how FaaS stymies distributed computing because there is no network addressability of serverless functions, two functions can work together *serverlessly* only by passing data through slow and expensive storage and how FaaS discourages open source service innovation since most popular open source software cannot run at the same scale as current commercial serverless offerings. Wang *et al.* [121] performed an analysis of the hardware differences of serverless platforms and their performance. This work has inspired some of the methodology of the experiments carried out in this section.

Despite the related work completed in the field, we have identified gaps in the knowledge that our work aims to fill. That is, we investigate potential interference effects on performance caused by load on the cloud platform by other users over the course of one month. This has resulted in the generation of a data set containing more than 500,000 function calls between the four platforms that we have published along with our benchmarking tools [122]. This work is performed

[1]https://aws.amazon.com/elasticbeanstalk/

on the current state of AWS Lambda, Google Cloud Functions, Microsoft Azure Functions, and IBM Cloud Functions in 2020.

The contributions of this chapter are as follows:

1. Analysis of effect configuration of serverless functions has on specification of the VM it executes on.

2. Investigation of how memory allocation affects function performance by recording various timing statistics and benchmarking metrics associated with function execution.

3. Investigation of potential interference from load on the platform, running our test system over the course of one month in order to detect any anomalies in runtime performance.

4.2 Methodology

The typical case for the invocation of serverless functions is through the use of REST APIs. For our tests, we deploy an observer virtual machine on each platform's traditional cloud computing offering. They were deployed on the same availability zone as the serverless functions in order to best nullify network latency. This observer's role was to invoke the functions via their API, record the request and response times, and finally push the results to an external MongoDB database. The observer script is written in Python and uses the *requests* package to trigger each function via its URL. We execute the observer script in two scenarios:

1. Two sequential function invocations for the definitive measurement of a cold and warm function start.

2. Fifty concurrent function invocations to prompt scaling and put increased workload on the function platform.

These scenarios were run every hour for one month. The serverless functions execute a number of routines that perform measurements on the specifications of the function's runtime environment. Such measurements include: function runtime, VM uptime, total available memory on VM, disk I/O throughput, CPU

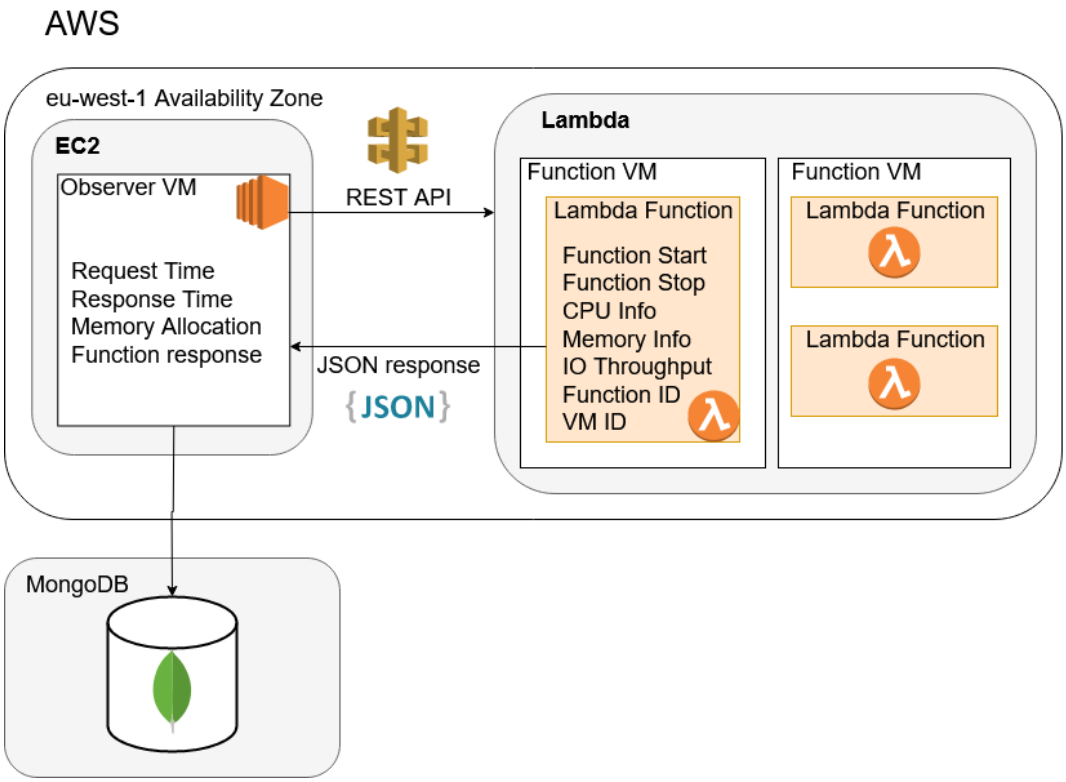

Fig. 4.1 AWS Experimental Setup

information, and unique identifiers for both function and VM instances. Much of these results were read from the Linux proc filesystem (procfs) where available.

We collate results of these routines into a JSON response that is sent back to the observer where it could append extra information such as: memory allocation of the function, total runtime of the function (request to response), and the start lag time (cold start time Section 4.4.1), before storing the results on the database. This process is illustrated in Figure 4.1 for AWS Lambda, and a nearly identical setup is used for the other platforms except using their equivalent cloud offerings.

4.3 Platform Architecture

In order to understand the architecture of the four platforms, we consult the official documentation for each platform in addition to running our aforementioned

Table 4.1 Recorded Attributes of Execution VM

Identification	
VM ID	A unique identifier for each execution VM. The method of measurement varies for each platform.
Function ID	A unique identifier for each function instance.
Previous Function IDs	A collection of Function IDs that have executed on a particular VM.
System Information	
CPU Speed	The speed in MHz of the CPU.
CPU Times	The amount of time in milliseconds a host has spent in user, system and idle time.
CPU Model	What CPU is used on the host.
Uptime	The timestamp the VM booted.
Total Memory	Total memory configured to the VM
Memory	The function memory allocation.

tests. Table 4.1 shows the measurements that we record using the function to gather information on the architecture of the platform.

Overview

AWS, Azure and IBM The proc file system exposes information about the execution VM that the functions run on and not just the function container. From our measurements, we found that the average VMs configured were as follows:

- **AWS** - Intel Xeon E5-2670 v2 processor @ 2.5 GHz with memory that varied based on the memory allocation of the functions executed on it such that it was slightly greater than the function's memory allocation
- **Azure** - Intel Xeon E5-2673 v4 @ 2.3 GHz with 2GB of memory
- **IBM** - Intel Xeon E5-2683 v4 @ 2.1 GHz with 16GB of memory

Google Cloud Functions Functions run in isolation on Google's proprietary hypervisor. This obscures the data we would have gathered from the proc file system to show mostly nothing. From our test, however, we could conclude that the average VM was configured with a Intel Xeon Skylake processor @ 2 GHz with 2GB of memory.

In contrast to AWS, Google and Azure simply allocate all their execution VMs with the maximum memory requirement for the largest function configuration of 2048MB and IBM allocate a larger memory allocation perhaps to handle more function containers per VM.

4.3.1 Unique VM Identifier

Uniquely identifying the VM used to execute the function allows us to gain a picture of the infrastructure each platform is using and can help answer questions about scaling, function isolation, disk access, etc. The work carried out by Wang *et al.* [121] determined that for AWS Lambda functions, the VM can be identified via the `/proc/self/cgroup` file, where an entry contains a unique identifier *sandbox-root-* followed by some random characters. For Azure, they proposed the use of an environment variable called *WEBSITE_INSTANCE_ID*. However, at the time of our experiments, no such variable could be found. For both Azure and IBM, the `/proc/machineid` file is used to identify VMs. No means to uniquely identify a Google Cloud Function's execution VMs was established. Due to the isolation Google's proprietary hypervisor imposes on the functions, access to the information normally stored in the proc file system is abstracted to show nothing, therefore, information to identify a machine could not be found. A heuristic to identify a VM by assuming that they all have unique boot times by Lloyd *et al.* [117] was also disproved by Wang *et al.* [121], deeming it unreliable. In order to gather meaningful data for comparison with the other platforms, we propose a method of identifying function containers by writing some *Unique Identifier* (UID) to a file into the `/tmp` folder. It was found that this file could be read by other functions that are executed in the same container. This was used, in conjunction with the boot time information as a sanity check, to ensure that the UID is indeed allocated to one container.

4.3.2 VM hardware differences

We examine the hardware differences from our data set of 563,276 function invocations. This exposed 156,856 VMs on AWS, 1392 VMs on Azure, 114 VMs on IBM and 21271 unique function containers on Google. It was stated by Wang *et al.* [121], and confirmed ourselves through correspondence with Google employ-

Table 4.2 AWS Lambda Function Memory to VM Total Memory

Function Memory (MB)	VM Total Memory (MB)
128	192.484
256	331.740
512	633.804
1024	1190.860
*1024	1717.196
2048	3230.668

ees, that Google Cloud Functions execute on their proprietary hypervisor that abstracts as much information about the execution VM as possible. However, we were successful in identifying function containers and could infer the CPU model information from the measurements we took. Two fields that were not hidden in the `/proc/cpuinfo` file were the CPU model id and the speed. We determine CPU models using the model number entry in the `/proc/cpuinfo` file, which gives a decimal number that can be converted to hexadecimal and then cross-referenced against the CPUID Signature table found in the Intel Architectures Software Developer's Manual, Volume 4 Chapter 2 [123].The CPU models found on each platform are collated in Table 4.3. All CPUs were a member of the Intel Xeon product line with IBM boasting the latest versions of such, comprising mainly versions 3 and 4. AWS has a more homogeneous CPU configuration than the other platforms, opting for 99.93% of them being Intel Xeon E5-2670 v2.

We also measure the total memory configured to the VM. In AWS, we observe that this value varied depending on the memory allocation of the function itself. A mostly consistent mapping was observed, as shown in Table 4.2, except for two outliers in the 1024MB functions, which ran on a VM with 1717MB of memory*. The other platforms had a constant memory configuration with Azure and Google allocating 2GB and IBM allocating 16GB.

4.4 Function Performance

The degree to which a developer can customise the configuration of a serverless function is quite limited. For AWS you can alter the memory allocation by increments of 64kB from 128MB to 3096MB and choose the timeout from a range of 1 second to 5 minutes. IBM has memory allocation in increments of 32kB from

Table 4.3 CPU Identification

Platform	Model ID (decimal)	Speed (MHz)	Model Name	Prevalence (%)
AWS				
	62	2500	Xeon E5-2670 v2	99.93
	62	3000	Xeon E5-2690 v2	0.07
Google				
	45	2600	Xeon E5-2670	24.12
	45	3300	Xeon E5-1660	0.02
	62	2500	Xeon E5-2670 v2	0.14
	63	2300	Xeon E5-2680 v3	4.79
	79	2200	Xeon E5-2650 v4	11.25
	85	2000	Xeon (Skylake)*	53.11
	85	2200	Xeon (Skylake)*	6.54
IBM				
	85	2300	Xeon Gold6140	19
	79	2100	Xeon E5-2683 v4	42.68
	79	2600	Xeon E5-2690 v4	18.83
	79	2200	Xeon E5-2650 v4	2.6
	63	2600	Xeon E5-2690 v3	9.79
	85	2100	Xeon Gold6130	7.05
	63	2000	Xeon E5-2683 v3	0.04
Azure				
	79	2300	Xeon E5-2673 v4	68.68
	63	2400	Xeon E5-2673 v3	22.07
	85	2600	Xeon Platinum8171M	9.24
			* Specific model could not be determined	

128MB to 2048MB and a timeout of 1 second to 10 minutes. Google allows you to choose from a predefined set of five memory allocations: 128MB, 256MB, 512MB, 1024MB and 2048MB. Finally, Azure does not allow the developer to configure their memory allocation opting instead for autoscaling memory. For our investigation of function performance, we use a series of measurements, as detailed in Table 4.4, on functions with memory allocations 128MB, 256MB, 512MB, 1024MB and 2048MB for AWS, Google, and IBM. For Azure, the same tests were run without the granularity of five memory allocations, i.e. only one function that was allowed to autoscale accordingly.

Table 4.4 Measurements of Function Performance

Total Runtime	Measured from time API invokes the function to the time a response is received.
Function Runtime	The time taken for a function to execute its tasks not including initialisation time of the function.
Start Lag	Function initialisation time. Measured from request time to main method start time.
CPU Utilisation	The number of primes a function can compute in 1000 ms.
Disk I/O	The I/O throughput of a function.
Number of VMs	The number of execution VMs created to handle scaling.

4.4.1 Cold Start Latency

One of the greatest problems facing serverless computing is the infamous cold start. Cold starts have been the subject of research papers and blogs [112, 124, 113, 110, 125, 126] and continue to be a major source of doubt for those considering a serverless-based application. They are the notable delays incurred when invoking a function for the first time. This is caused by the need for the platform to spin up a container that has all the required resources for a function to run. We measure the time taken from the function request to the time a function's main method began executing. We gathere these times for functions that were the first to execute on a new function container. This is determined by a file written to the `/tmp` directory that contains a log of all IDs of functions that had previously been executed in that container. The `/tmp` directory is ephemeral storage that lasts the lifespan of its function container. If the log was empty, it was a new container. We sample over a period of one month, running the tests every hour. This results in over 170,000 function invocations on AWS, Google, and IBM and over 36,000 on Azure (a fifth of the number of invocations since no separate tests for each memory allocation) for our analysis. The results are shown in Figure 4.2.

AWS Of 175,477 function invocations executed on AWS, 156,847 were cold starts. This would suggest that AWS may not prioritise the reuse of old containers. Of the five memory allocations chosen for the test function, 128MB had the slowest average cold start time at 346.73 ms. A point of interest is the near identical average values for the 256, 512, 1024 and 2048MB functions at 221ms ±3ms. It may be possible that AWS gives more precedence to these latter memory allocations.

Google Of the 175,357 function invocations executed on Google 21,253 were cold starts. There is high container reuse on Google Cloud Functions; some containers were hosting over a thousand function executions. A more expected stepping in results for each memory allocation's average cold start time was observed. These values were considerably higher than AWS with averages for 128, 256, 512, 1024 and 2048MB being 14465.52ms, 5722.33ms, 4681.37ms, 3689.48ms and 2865.49ms, respectively.

IBM Of 176,266 function invocations executed on IBM 37,820 were cold starts. Similar container reuse to Google is observed, although with much shorter cold starts: 2990.55ms, 1076.60ms, 1310.43ms, 1319.05ms and 915.49ms for the respective memory allocations. The results do not show a consistent trend which is discussed in Section 4.4.4.

Azure Of 36,176 function invocations executed on Azure, 1392 were cold starts. The average cold start time was 1997.63ms. The low number of cold starts is likely to do with there being only one function being invoked rather than five, thus allowing for greater container reuse.

From these results we can see the different tactics used by the different platforms. AWS opts for less container reuse (shorter lifespan), which results in more cold starts, although the cold start times were considerably shorter than the others. Google and IBM reused containers much more, thus minimising cold starts. This is especially useful for Google, whose cold starts were considerably higher.

4.4.2 Benchmarking - CPU Utilisation

Given a 1000ms time limit, check as many numbers as possible to determine whether it is prime. The method we use to check for prime numbers is trial division. For some number n across a set

$$1 < a \leq \sqrt{n} \tag{4.1}$$

n is prime if

$$\gcd n, a \geq 1 \tag{4.2}$$

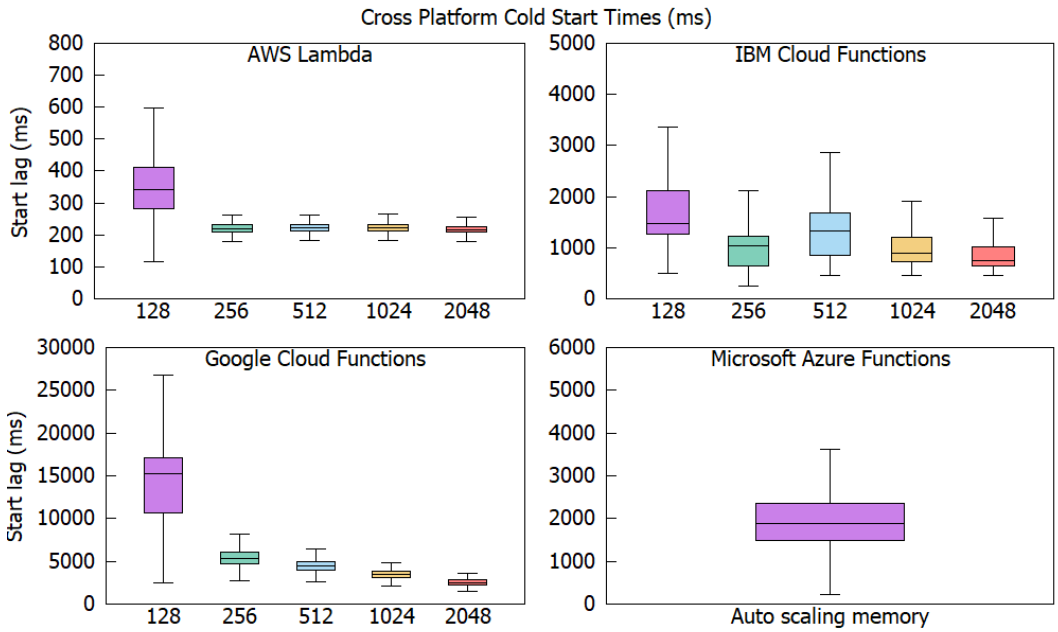

Fig. 4.2 Cross-platform cold start (request time to function start)

function $isPrime(n)$
  $start \leftarrow 2$
  $limit \leftarrow \sqrt{n}$
  while $start \leq limit$ **do**
    if $n < 1 \pmod{start}$ **then**
      return $False$
    else
      $start \leftarrow start + 1$
    end if
  end while
  return $True$
end function

This is a laborious way to calculate primes with the sole intention of consuming 100% of the CPU resources. The results are based on the volume of numbers checked for primality one by one (Figure 4.3).

AWS The volume of numbers that the function could check for primality increased as the memory allocation increased. AWS specifies that CPU power is distributed using time slicing, with larger memory allocations receiving longer time slices. The method of time slicing yields consistent access to the CPU as demonstrated by our results, i.e. the number of numbers checked had a low range of variation.

Google Average values were similar to those of AWS. However, each memory allocation had quite a wide range of values. This is likely due to the larger variety of CPUs used by the execution VMs, resulting in less consistent results.

IBM Average values were similar across all memory allocations with a smaller spread of values in the higher allocations. These values are comparable to the 1024MB functions in AWS and Google. There is a considerable spread in the lower memory allocations. However, we believe this is due to interference on the platform (discussed in Section 4.4.4).

Azure Again, a similar volume of primes was computed to the other platforms.

The similarity in the results suggests that the differences in topology (Table 4.3) are not necessarily a factor governing a function's CPU utilisation. The greater factor is likely the method that each platform employs for CPU resource allocation. Time-slicing is the most suitable method of allocating resources, as one can then use a more homogeneous back-end for function execution, reducing complexity. The algorithm for time-slicing will be the deciding factor in a function's CPU utilisation.

4.4.3 Benchmarking - Disk I/O Throughput

Serverless functions are more commonly associated with reading and writing to a proprietary storage solution: AWS Lamba → Amazon Simple Storage Service (S3) and Google Cloud Functions → Google Filestore. However, both offerings also allow for the reading and writing of temporary files to disk on the execution VM. Our disk throughput tests use the Linux *dd* command to read and write blocks of 512KB 1000 times and outputs disk throughput in MB/s (Figure 4.4).

AWS Throughput increased as memory allocation increased sharply after 128MB. However, it tapered towards the larger memory allocations, never getting above

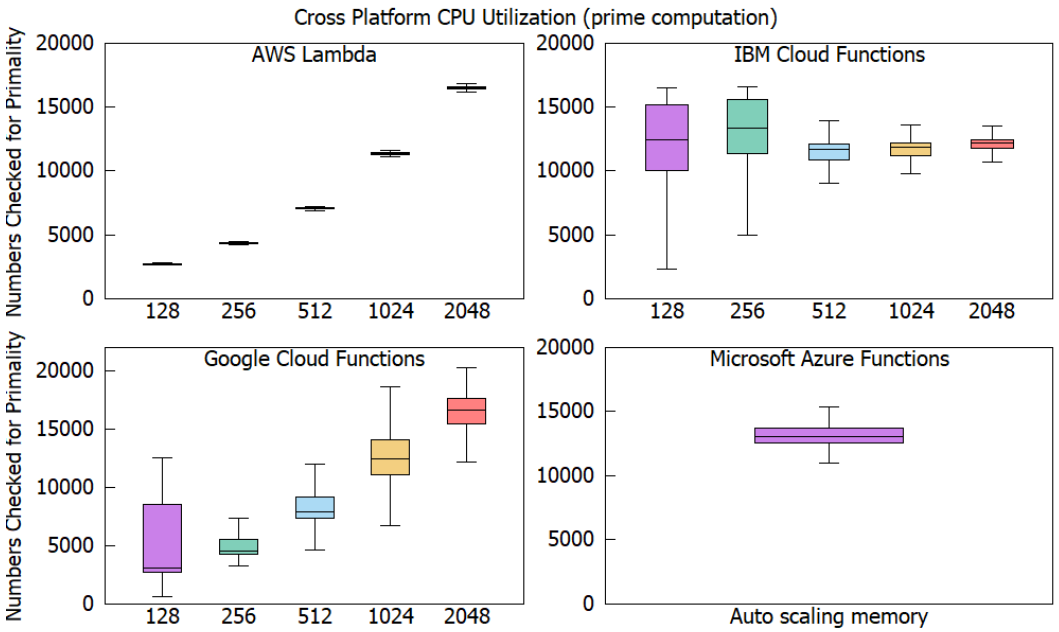

Fig. 4.3 Cross platform no. numbers checked for primality

3MB/s. Given the more homogeneous topology of the execution VMs observed, we can infer that this is the limit that is approached as the function is allocated more CPU time slices with a greater memory allocation.

Google A more varied range was recorded with an increasing throughput for greater memory allocations. This is likely related to the much larger number of CPU configurations (Table 4.3). Another recording of note is how much higher the values are to those of AWS. It is stated in Google Cloud Functions' documentation that these temporary files are actually stored in memory rather than on disk [127] which may be resulting in the greater throughput.

IBM Average values were consistent and approximately 0.6MB / s for each memory allocation. This, along with the previous test, further suggests that IBM Functions are not affected by memory allocation.

Azure The throughput recorded for Azure was the lowest of all platforms, averaging under 0.5MB/s. Disk throughput is important for more complex functions that depend on temporary memory access. As functions are billed on execution time, lower throughput may result in increased run time.

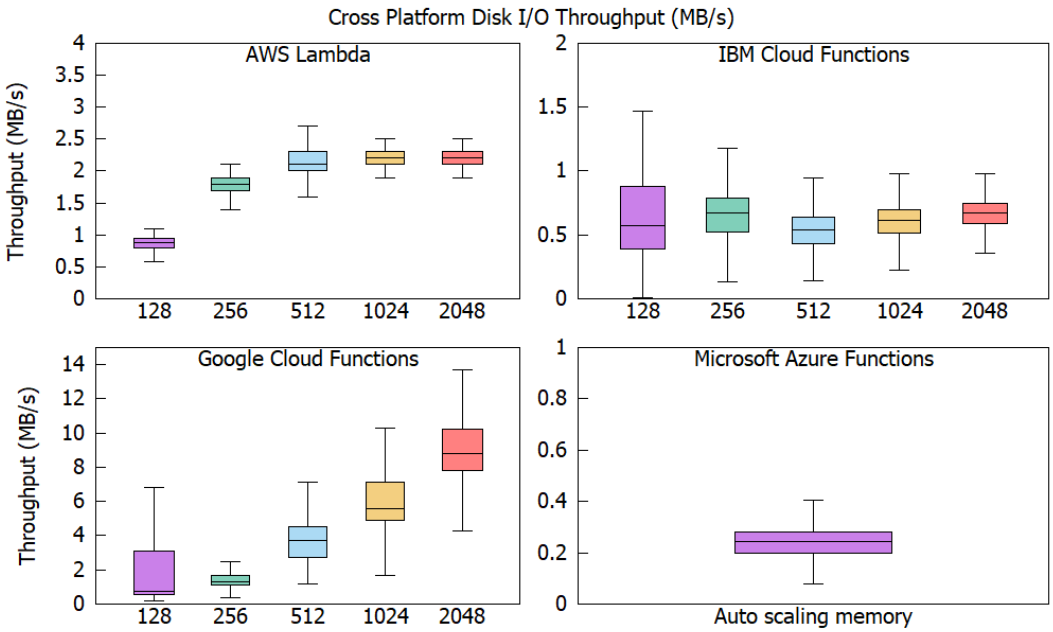

Fig. 4.4 Cross-Platform Disk I/O Throughput

4.4.4 Interference Effect on Performance

Serverless platforms employ a similar strategy in how functions are executed. A function runs within a function container on an execution VM within a predefined region of the serverless platform's cloud. The execution VMs are isolated from other individual users [121]. However, different users' VMs can exist in the same region or on the same host, potentially leading to interference effects. Other events such as wide-scale updates, system outages etc. may also interfere with the execution of the function. We ran our tests over the course of one month to investigate the potential effect of any interference that causes anomalies or predicable fluctuations in performance. We examine the effect on function run time, CPU utilisation, and disk I/O throughput. The results are a smoothed graph of data points taken from each hourly run of our test functions using gnuplot's *acspline* [128] for run time (Figure 4.5), CPU utilisation (Figure 4.6) and disk I/O throughput (Figure 4.7). Tables 4.5, 4.6 and 4.7 shows the observed variability, including the coefficient of variation for each platform.

AWS Running in region *eu-west-1*, we observe minimal variance in values for CPU utilisation. However, a sinusoidal pattern is visible in the results of the effect

on run-time and disk I/O throughput. The pattern becomes clearer with higher memory allocations. We believe this is due to higher memory allocations having greater access to the CPU (via time slicing) and, as such, will execute in a more consistent manner, therefore amplifying the visual effect of interference when it occurs. The pattern has peaks at 12:00 pm and troughs at 12:00 am each day, corresponding to higher use during the day than at night. This may be evidence to support the potential effect of regular load on the system.

Google Running in region *us-central1-a*, more erratic values were observed. However, consistent peaks and troughs are visible, similar to those of AWS. Unlike AWS, Google Cloud did not have a consistent topology for its execution VMs (Section 4.3.2), which may be the cause of the varied array of results. AWS Lambda functions ran predominantly on the same CPU configurations, which means that their results were more consistent, allowing us to see clearer interference patterns.

IBM Running in region *Dallas*, clear signs of strain on the platform were observed with large performance dips occurring at the beginning and near the end of our month of testing. There were a number of issues affecting IBM's cloud during this time according to their status history page[2], which may be the culprit. Another possibility is that IBM's serverless platform is not capable of consistent performance like its competitors, leading to anomalies like the ones captured by our tests.

Azure Running in region *Central US*, we observe results similar to Google to be erratic in amplitude but periodic in time peaks and troughs in its performance. Our month-long analysis demonstrates that there is interference on each platform and we propose potential sources for such interference. As serverless functions are billed per 100ms, we can see that the fluctuations observed from our tests would have an effect on the final cost. As well as cost, there are also considerations to be made for more critical functions requiring consistent performance.

[2]https://cloud.ibm.com/status?selected=history

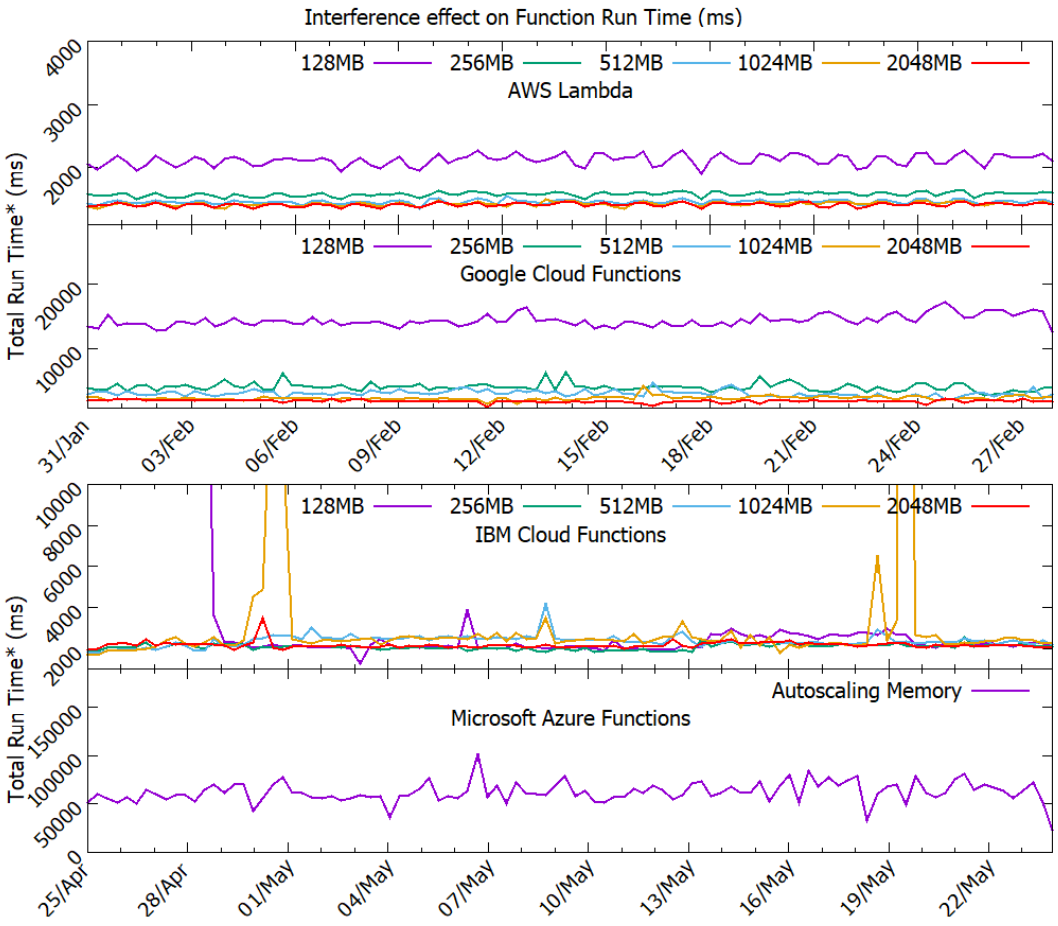

Fig. 4.5 Function run times over one month *Smoothed hourly values. Note: IBM functions peak off-chat with a value of 66978ms

4.5 Encountering Denial of Wallet

In the process of conducting these tests, we unintentionally perform DoW on ourselves. Early implementations of our tests had all experiments running one after another on the same function and that function being executed 1000 times an hour to induce automatic scaling. While we had calculated that this would not consume our free allowance of 1 million function invocations on AWS Lambda, we did not account for the compute limit of 400,000 GBs. As all the experiments were being executed, some functions took upwards of 20 seconds to complete, which when deployed on a 2048MB Lambda function cause the free compute limit to be reached within a few hours.

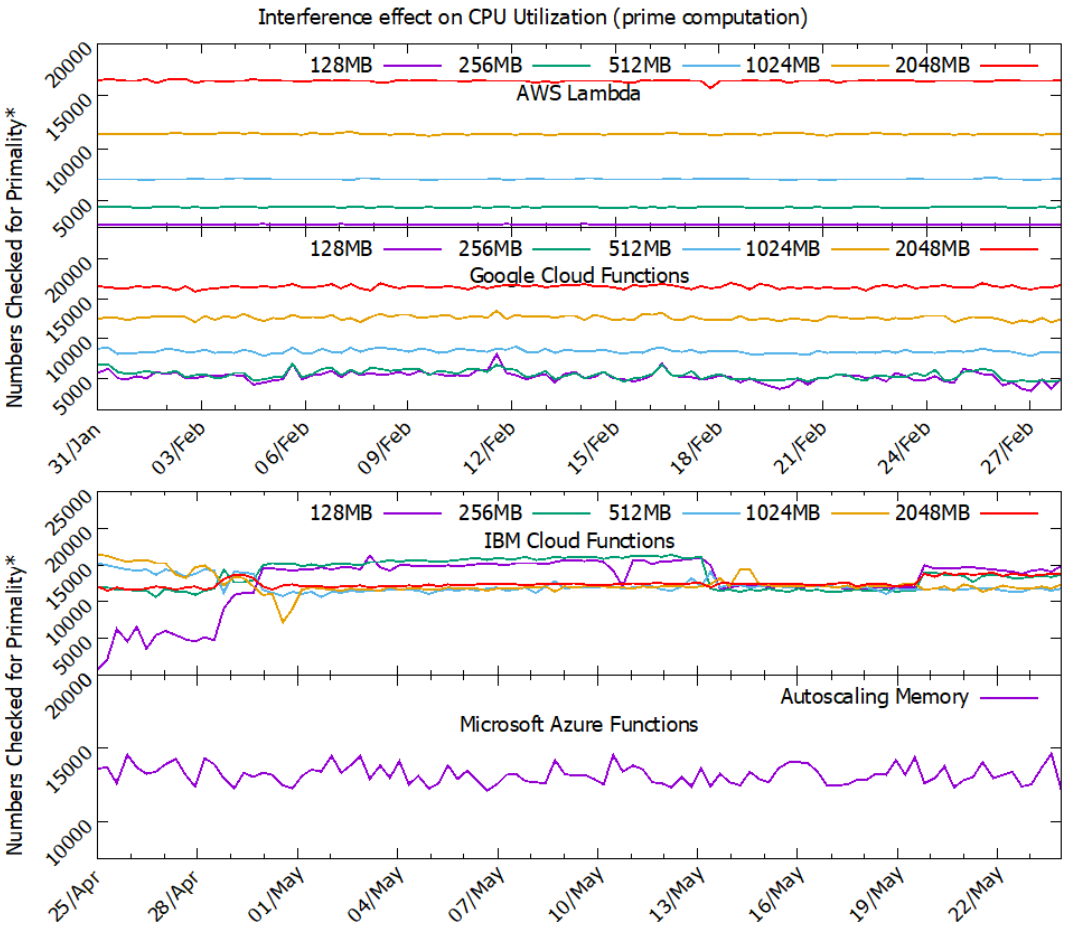

Fig. 4.6 Function CPU Utilisation (prime computation) over one month *Smoothed hourly values

Action was taken within an hour of the issue being discovered, thankfully resulting only in $40 of damage. However, it was at the expense of having to shut down the entire experimental setup and delaying the data gathering by a month.

This effect, as previously discussed, was known as EDoS in academia or, more recently, the term DoW has been adopted within the wider cloud computing community. It is from this experience that we decided that a more detailed investigation was required on the potential for this self-inflicted disaster to be weaponised as a targeted attack.

Beyond the request spamming DoW, the information presented in this chapter would have been accessible to any attacker previously. Such data can inform an adversary on potential attack strategies in order to leverage serverless function

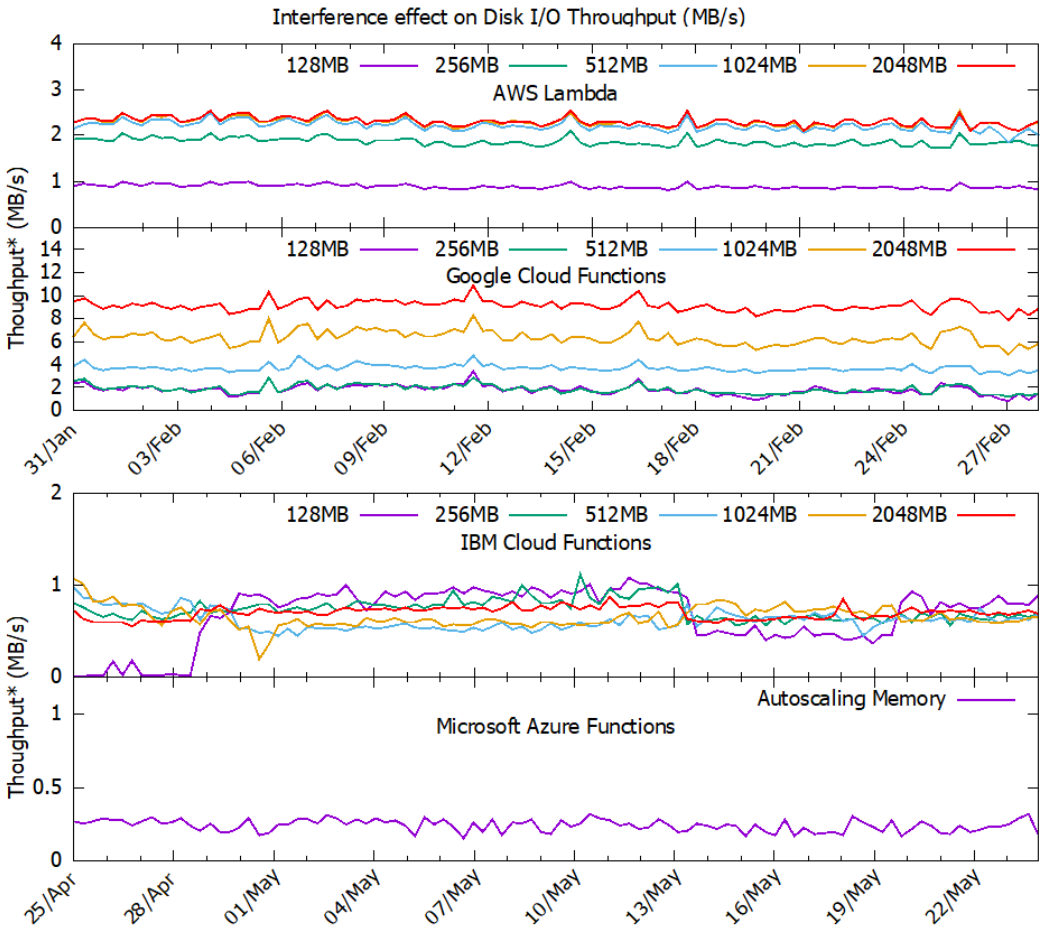

Fig. 4.7 Function Disk I/O Throughput over one month *Smoothed hourly values

exploitation. While such exploit identification is beyond the scope of this research, we believe that it is important to identify potential venerability points in the early stage of our investigation on DoW.

4.6 Summary

In summary, this chapter presents an in-depth analysis of the underlying hardware differences of the four biggest commercial serverless platforms and the effects on performance for differing memory allocations. We found that AWS has a different approach to function container reuse that we theorise to be a design choice to address scaling and cold start issues. We determined that memory allocation largely affects the performance of serverless functions on these platforms and

Table 4.5 Observed variability on each platform for function runtime

Platform	Function Memory (MB)	Standard Deviation (ms)	Mean (ms)	Coefficient of variation
AWS				
	128	126.04	2214.69	5.69
	256	71.72	1612.41	4.45
	512	76.36	1494.12	5.11
	1024	76.15	1458.08	5.22
	2048	74.72	1445.36	5.17
Google				
	128	5686.66	14452.68	39.35
	256	1854.73	4223.67	43.91
	512	1532.46	3188.12	48.07
	1024	1362.32	2528.19	53.89
	2048	1157.38	2088.89	55.41
IBM				
	128	8661.12	5177.24	167.29
	256	1141.66	2244.39	50.87
	512	850.62	2506.45	33.94
	1024	3061.52	2792.65	109.63
	2048	1008.78	2181.85	46.24
Azure				
	n/a	44913.14	62160.29	72.25

must be a serious consideration when developing applications. Finally, we performed a month-long observation on function performance, creating a large data set of function benchmarks and uncovering potential interference effects due to increased load on the platform during the day and strain caused by maintenance on the platform. Through the process of this investigation, we came face-to-face with the potential threat of DoW. It was from this accident that we decided to focus on the largely under-research attack. The information in this chapter is not easily discovered from the cloud providers themselves. As such, we believe that by publishing our findings we are helping developers inform themselves on potential avenues adversaries my utilise in order to conduct attacks.

Table 4.6 Observed variability on each platform for CPU utilisation

Platform	Function Memory (MB)	Standard Deviation (numbers)	Mean (numbers)	Coefficient of variation
AWS				
	128	40.92	2717.97	1.51
	256	56.28	4374.51	1.29
	512	76.11	7079.63	1.08
	1024	112.87	11385.87	0.99
	2048	186.85	16474.25	1.13
Google				
	128	3149.86	5150.98	61.15
	256	1992.45	5427.00	36.71
	512	1355.52	8358.26	16.22
	1024	1892.16	12588.69	15.03
	2048	1659.63	16466.22	10.08
IBM				
	128	3697.85	12010.97	30.79
	256	2630.06	13043.96	20.16
	512	1945.07	11384.95	17.08
	1024	1740.70	11845.25	14.70
	2048	967.96	12226.08	7.92
Azure				
	n/a	876.89	13189.77	6.65

Table 4.7 Observed variability on each platform for disk I/O throughput

Platform	Function Memory (MB)	Standard Deviation (MB/s)	Mean (MB/s)	Coefficient of variation
AWS				
	128	242.26	783.32	30.93
	256	7.40	1.87	394.71
	512	9.88	2.28	434.17
	1024	7.74	2.31	334.85
	2048	5.23	2.28	229.51
Google				
	128	318.15	379.37	83.86
	256	274.38	106.59	257.42
	512	1.28	3.68	34.66
	1024	2.15	6.32	34.03
	2048	1.83	9.08	20.14
IBM				
	128	0.31	0.60	52.43
	256	0.18	0.66	27.29
	512	0.17	0.54	31.17
	1024	0.16	0.60	27.13
	2048	0.12	0.66	18.83
Azure				
	n/a	0.05	0.24	21.73

CHAPTER 5

Synthesis and Simulation - Continuing Research in the Absence of Data

The greatest difficulty of this research topic is analysing DoW on the specific domain of serverless computing. Despite similar types of attack, such as, DDoS and EDoS, there is an absences of data on serverless computing instances of DoW. To this end, a great deal of work was put into the development of a means to safely test the effects of the attack that would not deplete the research funding. Also, a method of synthesising data for future work on training detection systems in the absence of existing historical attack data was developed, thus answering *RQ2* "How can a lack of historical data be overcome in the preemptive investigation of this attack, where traditional sandbox testing is not viable?".

This chapter details the proposed solutions to these issues. Content from **Kelly, D.**, Glavin, F.G., and Barrett, E. (2022) "DoWTS – Denial-of-Wallet Test Simulator: Synthetic Data Generation for Preemptive Defence" Journal of Intelligent Information Systems [129] details the development of the *Denial of Wallet Test Simulator (DoWTS)* tool that allows us to quickly generate synthetic traffic logs of normal usage and attack data. Also outlined is an isolated testbed from **Kelly, D.**, Glavin, F.G. and Barrett, E. (2021) "Denial of wallet - defining a looming threat to serverless computing" Journal of Information Security and Applications [83] that

enables safe deployment of an analogue serverless environment for analysis of attack and mitigation strategies on deployed serverless applications.

5.1 Introduction

In this chapter, we present our work on a tool for generating synthetic usage data on a serverless application, called Denial-of-Wallet Test Simulator (DoWTS) and an analogue serverless platform for real-time testing of attacks. Chronologically, in this research, DoWTS was developed after the serverless platform analogue. However, we will present it first as it reflects how the two tools are utilised later in Chapter 6 for training and testing of a DoW mitigation system. Namely, the contributions discussed in this chapter are:

1. Creation of DoWTS as a means of generating normal usage data in the absence of historical data. We detail extensively our heuristics for generating such data and perform an in-depth evaluation of the validity of our synthetic "normal" usage generator. Such validity checks include visual analytics, quantitative comparison, and a number of statistical tests used for assessing the synthetic data similarity to real data. We surmise three basic attack vectors that will serve as an initial investigation into how an adversary may conduct DoW attacks.

2. Design of a serverless platform analogue that serves as a testbed to demonstrate the validity of future mitigation strategies. We describe the design and setup of this system for ease of replication by the research community.

5.2 DoWTS - Denial-of-Wallet Test Simulator

In order to gain insight into and tackle related cybersecurity threats that target public facing applications, the inspection of network traffic is required. However, such data can raise privacy concerns as network traffic logs may reveal personal information of users such as IP address, browsing data, and in the cases of ecommerce data; purchase history. Therefore, it can be difficult to obtain realistic data for the training of mitigation systems. There are additional factors that increase

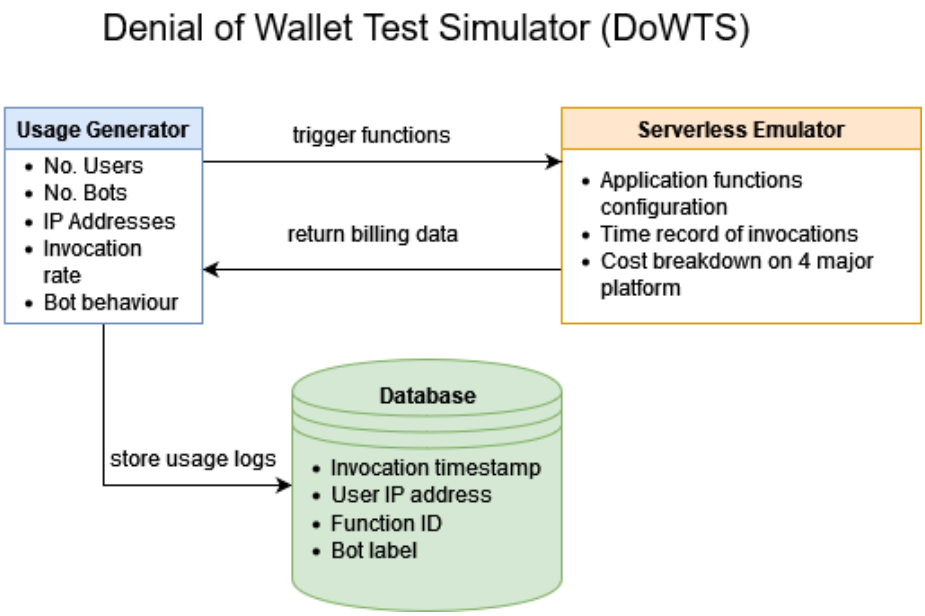

Fig. 5.1 DoWTS usage flow

the difficulty in obtaining the required data for training serverless application-based threat detection systems. In general, there is a lack of available traffic data that log interactions with function triggers that make up a serverless application. As the serverless paradigm undergoes continued adoption, this lack of data will no doubt be remedied. However, in this early stage of serverless specific security research, it remains a large issue. An additional factor that makes DoW research specifically more difficult is that this attack targets the capital of application owners. As such, it is a costly and tedious ordeal to investigate DoW on *in-the-field* deployments of serverless applications due to the requirement to literally pay for any investigations into various attacks. The solution to these issues is to utilise synthetic data generators to create the required datasets for further use in mitigation system training.

5.2.1 Architecture

DoWTS is a simulated serverless platform that will emulate the cost damage of DoW and create pseudo timestamped datasets of request traffic [130]. DoWTS generates synthetic baseline request traffic via a system of heuristics modelled on existing datasets of requests on e-commerce websites [131, 132]. This serves

as background usage upon which any number of theorised DoW attacks can be launched. DoWTS calculates the current cost of serverless function invocations per simulation time step for four of the largest commercial platforms. It also generates usage log data for every function invocation with a label that denotes whether it was a bot or legitimate traffic. Such labelled data will be used in future research on classifying legitimate traffic.

DoWTS is written in Golang[1] in order to take advantage of the multi-threading capabilities of *goroutines*. It is composed of three main components; Serverless Platform Emulator, Usage Generator and a MongoDB database (Figure 5.1).

DoWTS is operated through interaction with the usage generator as shown in the sequence diagram in Figure 5.2. The user specifies the combination of attack method and user identification spoofing according to the attack traffic generator. The scenario is programmed with predefined values for the parameters listed in the synthetic traffic generator. Finally, the scenario is executed.

DoWTS creates log files for each of the simulated platform's cost, compute time, and number of function invocations per run. This is to accompany the generated dataset of traffic to the serverless functions.

Serverless Platform Emulator

We developed the Serverless Platform Emulator in order to compare the effects of Denial-of-Wallet across multiple commercial platforms. It includes go-packages each for AWS Lambda, Google Cloud Functions, Azure Functions, and IBM functions. Utilising the openly available pricing formulae for each platform, the emulator keeps count of: total cost, total invocations, and total runtime of the functions on that platform. DoWTS can be used to model the expected expenditure on the serverless components of an application across the four largest platforms. This information is useful in the decision making process for which platform to choose as the host of the serverless functions. However, while our simulator does indeed offer this utility, its primary purpose is the generation of datasets for DoW detection research and as a means to safely study a variety of attack vectors. DoWTS can be configured with specific instances of serverless applications by varying the input parameters, such as usage metrics and function configuration.

[1] https://go.dev/

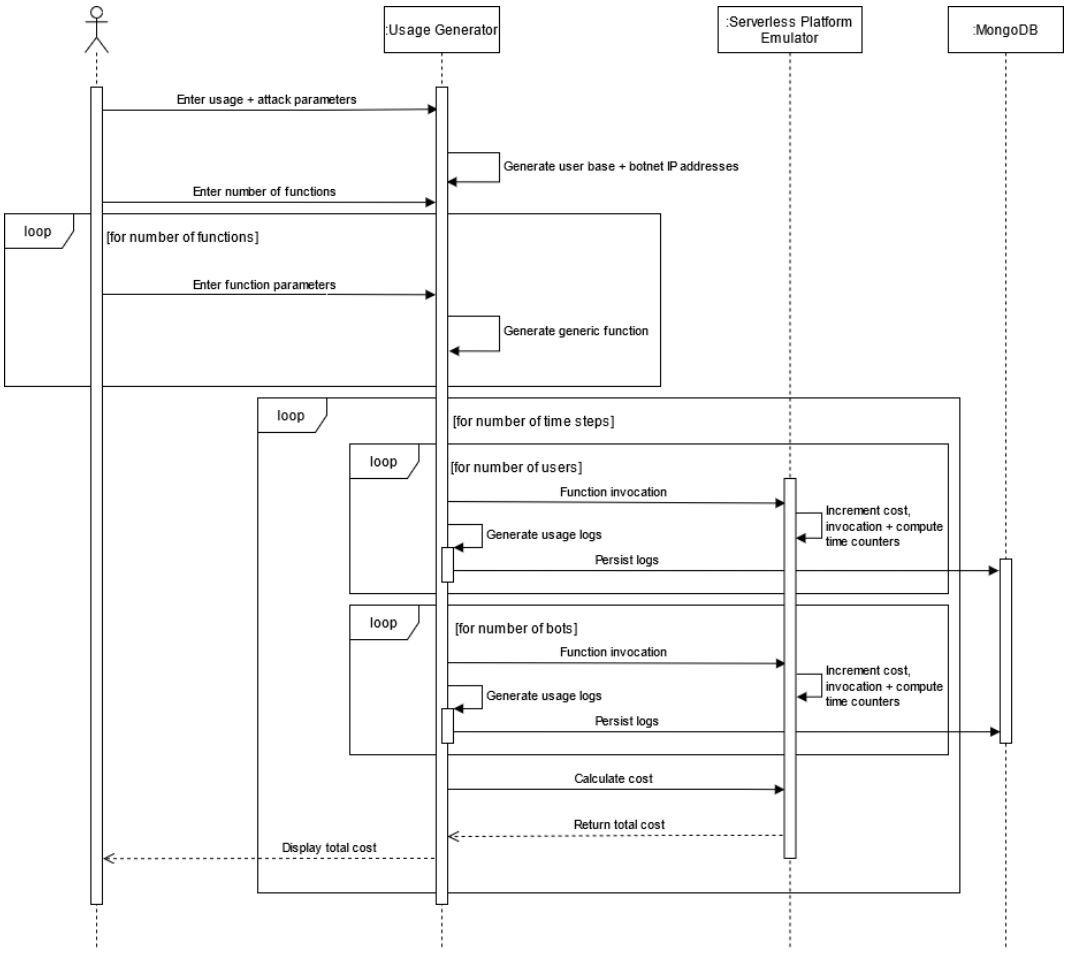

Fig. 5.2 Sequence diagram of DoW attack simulator and dataset generator

5.2.2 Usage Generator

The usage generator comprises an attack traffic generator, synthetic traffic generator, and dataset-based traffic generator. The role of the usage generator is to manage the timings of function invocations to the Serverless Platform Emulator, log the function invocations, and persist them to the database. All three generators have a standardised output format for each pseudo requests comprising: *timestamp, IP address, function ID* and a label denoting whether the request came from the attack generator or not.

Dataset Based Traffic Generator

There is a lack of historical data on both attacks and general usage on serverless applications available to the public. However, it is feasible to use certain datasets that capture usage on traditional web applications where the endpoints of an interaction are recorded. For example, the following datasets [131, 132] are request event histories recorded over five-month periods each. The data was collected via Open CDP [133] on a fashion and an electronics e-commerce store, respectively. These datasets contain a field *event_type* that corresponds to what URL endpoint a user interacted with. If this is to be translated into a similar mechanism on serverless applications, these can be interpreted as API triggers to serverless functions. As such, the recorded traffic, in these datasets, can be streamed via the dataset-based traffic generator as the legitimate traffic in the DoWTS system. The result of translating this traffic to the DoWTS system allows for the fine-tuning of our synthetic data generator.

Synthetic Traffic Generator

The synthetic traffic generator takes in a number of parameters from the user, in a given usage scenario, that define:

- **User base size** - The total number of users that access an application
- **Botnet size** - The number of bots attacking the application
- **Time step** - The granularity of time for the simulation run (seconds, minutes, hours, days, weeks, months). This will affect the data logs generated
- **Number of time steps** - The length of the simulation
- **Users per time step** - The number of users accessing the application in a given time step
- **Requests per time step** - The number of function requests in a given time step

Once these parameters have been entered, DoWTS generates IP addresses for the users and botnet. The user is then prompted to enter in the configuration of however many functions are present in their application i.e. memory allocation

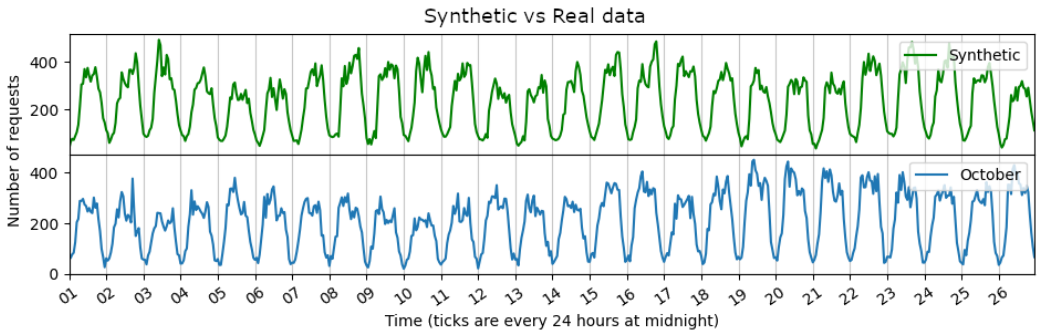

Fig. 5.3 Synthetic data traffic pattern compared to example of real traffic pattern (e-commerce shop 2 in October).

and function execution time. The functions may then be grouped into chains of functions that logically execute one after another. These chains contain a parameter that allows for the distribution of traffic to certain chains more than others for instances where there are functions that are more commonly used than others.

Traffic Synthesis The timing of requests is dictated by a system of heuristics devised in reference to the output of the e-commerce datasets [131, 132]. The hourly count of requests is used as the metric on which we base these heuristics. The full methodology of our data synthesis is presented in Appendix A. In summary, our method uses a Poisson distribution for underlying random fluctuations in traffic and a number of features are coded into the traffic such as:

- Weekly traffic peaks
- Large dips in traffic every night
- Noise during those dips
- Noisy peaks in traffic during the day
- Day time peaks longer than night time dips

Figure 5.3 shows the created synthetic traffic pattern compared to a sample of real traffic.

Attack Traffic Generator

There are no documented instances of targeted DoW on serverless to date. Therefore, it is required that our research take on the role of devising potential attack patterns that may cause DoW in an effort of preemptive defence. This research focuses primarily on presenting DoWTS as a system for future attack research. As such, the attack generator posed here serves as a proof of concept. We surmise three rudimentary *leech* attacks [83] to demonstrate DoWTS's capability in generating attack data. We have modelled three potential attack vectors for a simple long-term leech. These attacks take precedent from traditional DoS attacks but also aim to highlight the unique capabilities of a leech for DoW.

1. Constant rate – eg. 2000 requests per hour (rph) per bot
2. Geometric rate – start low (eg. 10 rph) then increase by factor (eg. 1.001) of rph
3. Random rate – attack with randomly varying numbers of requests

On top of the three attack vectors, DoWTS also implements a number of methods of recycling and changing the IP addresses of the botnet to either emulate a changing botnet or spoofing IP address. These methods are as follows:

1. No IP address change
2. Change bot IP address every x number of requests
3. Change bot IP address every x amount of time steps
4. Pool bot IP addresses for reuse over time

These serve as a starting point from which we begin the challenge of training classifiers to detect varying attack strategies for DoW (Chapter 6). Initial attack vectors will be simplistic, but over time we will investigate increasingly difficult patterns to detect, utilising DoWTS to generate data.

5.2.3 Execution Example

The configuration parameters for DoWTS can be arbitrarily chosen or selected in order to replicate a known serverless application. In this example, we choose

the usage values described in an AWS serverless application load testing guide [2]. It describes a geolocation question and answer based application along with its expected traffic load. From this we can set the following parameters for normal operation:

- **User base size** - 1,000,000
- **Time step** - 1 hour
- **Number of time steps** - 730 (1 month)
- **Users per time step** - 1,500
- **Requests per time step** - 61,000

The *time step* and *requests per time step* are the most important values in this example as they set the expected load on the application. The usecase also provides information on how functions should be chained (Figure 5.4). These are the four API requests a user can make when interacting with the application (GET/POST Questions and Answers). Each API request triggers a chain of Lambda functions. For the purpose of DoWTS, we are only interested in the number of Lambda functions and in what order they execute. Understanding the mechanism of the example application is not required to generate traffic data with DoWTS.

From this we can infer the following function chains:

1. **POST Question:** *PostQuestions* → *ProcessQuestion* → *Publish*
2. **POST Answer:** *PostAnswers* → *ProcessStarAnswers* `or` *ProcessGeoAnswers* → *Aggregation* → *Publish*
3. **GET Answer:** *GetAnswers* → *Aggregation* → *Publish*
4. **GET Question:** *GetQuestions* → *Publish*

Also, we know the expected load on each endpoint, so we can set the factor on each function chain that determines the ratio of the total traffic it should receive.

Running DoWTS on this configuration gives us a result for normal operation usage in this scenario. We can visualise the traffic by counting the requests per hour (Figure 5.5).

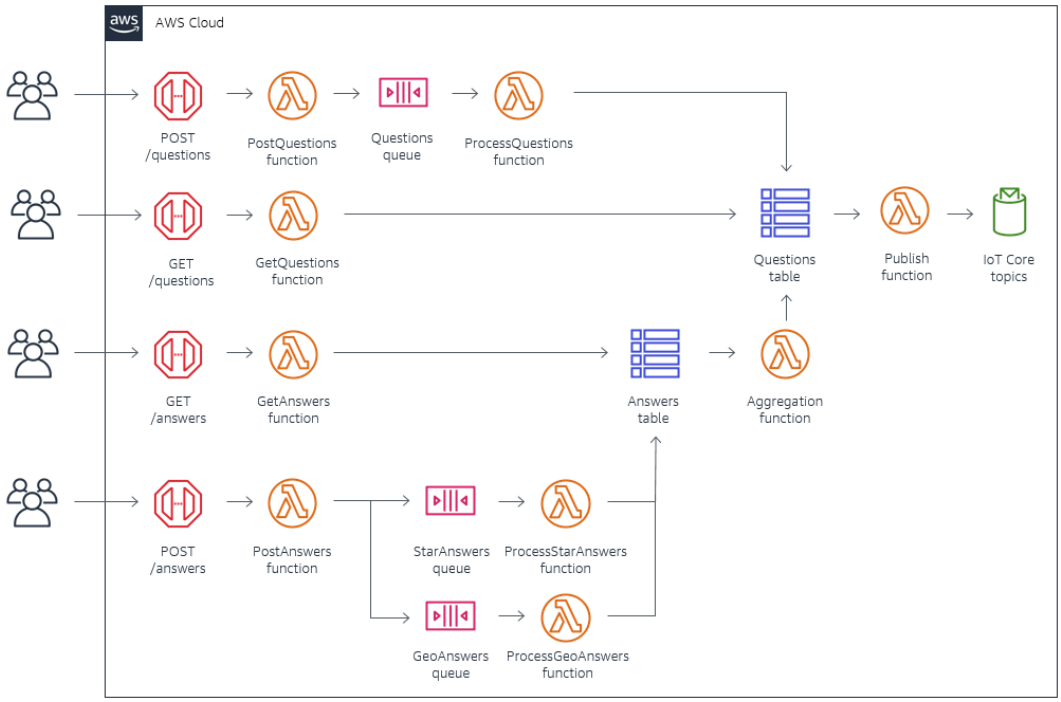

Fig. 5.4 Architecture diagram of application in AWS load testing example [2]

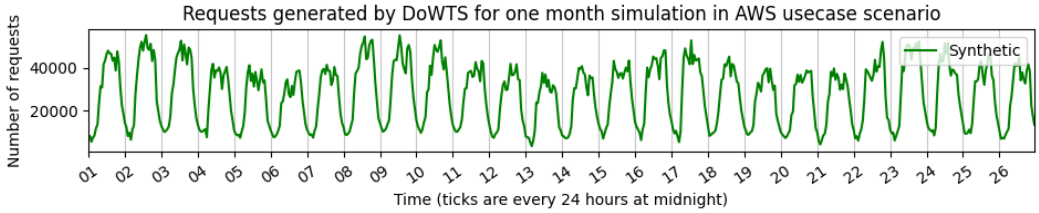

Fig. 5.5 Normal traffic on usecase application

This configuration is used as the background traffic upon which we can perform three additional runs with each of the attack patterns activated. The first run will be a continuous-rate leech attack of 100 bots performing 200 requests each throughout each hour. This is a very low rate that would not trigger any sensible rate-limiting rules on a WAF. The second run will be an geometric rate leech attack of 100 bots that will start at 10 requests each and increase by a factor of 1.005 each time step. The goal is to lull the application owner into thinking that they are receiving a steady increase in legitimate traffic. Finally, a random rate leech attack of 100 bots will be performed by randomly selecting a number of requests to send from 0 to 400. Figure 5.6 visualises the effect on traffic these attacks have.

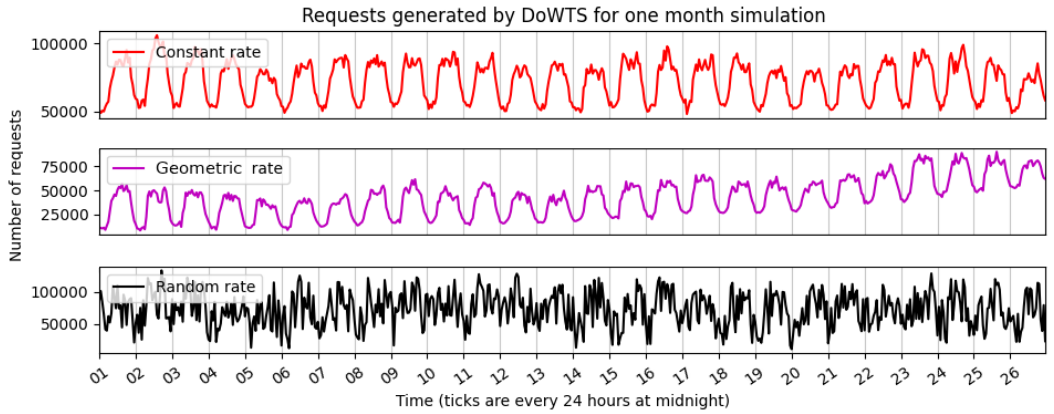

Fig. 5.6 Three leech attack variants on usecase application

Table 5.1 Operational cost of the application in each scenario for one month of each serverless platform

	AWS	Google	Azure	IBM
No attack	$20.69	$26.62	$20.02	$17.15
Constant rate	$77.40	$85.92	$74.71	$68.44
Geometric rate	$49.49	$56.65	$47.78	$43.25
Random rate	$79.18	$87.74	$76.42	$70.07

DoWTS also calculates the cost of these function invocations. Table 5.1 shows how attacks increase the operational costs of the application through the end-of-the-month bill. Even at low scale we can see that leech attacks may double the costs.

5.2.4 Synthetic Normal Data Evaluation

A major challenge with regard to synthetic data generation is to ensure that the generated data correlates closely with the real world data as much as possible. A simulation platform is only useful if it can emulate the system it models and give confidence that any classifiers trained using the simulation data will be robust when deployed in the real world. To assess the usability of DoWTS in future work on training classification algorithms, we must evaluate the validity of the data produced by our synthetic usage generator. We used the SynthGauge [134]

framework for performing a statistical analysis and also conducted our own visual analysis of the data.

SynthGauge is a Python library that provides a framework for evaluating the utility and privacy of synthetic datasets, using a range of metrics and visualisations. It is the result of a collaboration between the UK Office for National Statistics and The Alan Turing Institute. Our evaluation focuses on the timings of arrivals of requests, from which we bin traffic to achieve an hourly count of requests. On the basis of this data, we compute the following to assess the validity of our synthetic data.

- **Quantitative Comparison** - calculation of useful statistical values such as: Mean, Standard Deviation, Min/Max, Interquartile Range, and Median. These values are useful for an initial comparison of real and synthetic datasets in order to determine whether they are similar in scale (number of data entries).

- **Kolmogorov Smirnov (KS) Test Statistic** [135] - calculates the maximum difference between the cumulative distribution functions of the traffic count in the real and synthetic datasets. If the returned statistic is small, then it can be said that the datasets come from the same distribution.

- **Wasserstein Distance** [136] - or Earth Mover's distance, can be thought of as calculating the amount of work required to move from the distribution of the synthetic data to the distribution of the real data. The distance is zero if the distributions are identical, and increases as they become less alike. We compare the Wasserstein distances between multiple months of real data and compare the distance between real and synthetic data with that.

- **Jensen Shannon Divergence** [137] - describes the difference between the real and synthetic distributions of the traffic count in terms of entropy. We can think of the Jensen-Shannon divergence as the amount of information, or entropy, encoded in the difference between the real and synthetic distributions of the traffic count. The distance is zero if the distributions are identical, and is bounded above by one if they are nothing alike.

For consistency in comparison, the range of data for each month was reduced to the shortest month of data gathering to accommodate the shortest

Table 5.2 Quantitative comparison of dataset 1 on the number of requests per hour.

Month	Mean	StD	Min	25%	50%	75%	Max
October	5544	2824	673.0	3010	6081	7492	15201
November	6245	3601	694.0	2970	6626	8212	23257
December	5131	2555	675.0	2702	5649	7126	10912
January	5533	2858	415.0	2637	6054	7902	11473
February	5986	3011	94.0	3136	6672	8429	14427

Table 5.3 Quantitative comparison of synthetic data based on dataset 1 on number of requests per hour.

	Mean	StD	Min	25%	50%	75%	Max
Synthetic	5415	2685	771	2578	5937	7633	10750

period of measurements in the dataset. Henceforth, the fashion and electronics e-commerce datasets shall be referred to as *dataset 1* and *dataset 2* respectively.

Quantitative Comparison

Our first measure of validation is a comparison of quantitative metrics for each dataset. We compute the mean, standard deviation, minimum value, maximum value, interquartile range, and median of the number of requests per hour. By computing these values on each of the five months of real data in both real datasets we can determine a range of values that should constitute acceptable readings for our synthetic data (Table 5.2 and Table 5.4). This will suggest that the combination of input parameters to DoWTS, in conjunction with the synthetic usage generator heuristics, will output traffic in line with expected values based on historic readings. Table 5.3 and Table 5.5 confirm that the synthetic data falls within an acceptable range of the real data it is based off.

Statistical Testing

Our next measure of validity is a number of statistical tests chosen to determine the similarity between the real and synthetic datasets. Our quantitative comparison confirmed that our datasets contain similar values. As such, we can apply the three aforementioned tests to determine whether they are distributed in a similar

Table 5.4 Quantitative comparison of dataset 2 on the number of requests per hour.

Month	Mean	StD	Min	25%	50%	75%	Max
October	210	107	21	104	225	296	450
November	270	134	0	133	299	380	548
December	208	101	8	110	236	289	428
January	253	122	23	130	280	354	545
February	251	1115	30	136	284	339	460

Table 5.5 Quantitative comparison of synthetic data based on dataset 2 on the number of requests per hour.

Month	Mean	StD	Min	25%	50%	75%	Max
Synthetic	235	118	33	111	260	331	494

Table 5.6 Statistical evaluation of dataset 1 on the number of requests per hour against itself to establish baseline values.

	KS Test statistic	p-value	Wasserstein Distance	Jensen Shannon Divergence
Oct - Nov	0.1202	0.0002	730.4824	0.0264
Nov - Dec	0.1875	<0.0001	1113.7660	0.0546
Dec - Jan	0.1218	0.0002	427.7308	0.0218
Jan - Feb	0.0946	0.0075	464.5721	0.0091

manner. These tests were first performed on each month of real data on its following month (Table 5.6 and Table 5.8). These values are used to confirm the validity of the tests, as we know that these datasets are similar. Therefore, if our synthetic dataset is equally similar, then it would pass as legitimate traffic. The suite of tests are run against each month of each real dataset and the synthetic dataset (Table 5.7 and Table 5.9). The results indicate that the generated data is suitably distributed when compared to the real dataset on which it is based.

5.2.5 Visual Analysis and Evaluation

Finally, a sanity check is performed through visual analysis of traffic data. The traffic count is plotted with respect to time invoked for all datasets and grouped for visual comparison. Figure 5.7 and Figure 5.8 clearly show obvious day/night

Table 5.7 Statistical evaluation of synthetic data against dataset 1 on number of requests per hour.

	KS Test statistic	p-value	Wasserstein Distance	Jensen Shannon Divergence
Oct - Synth	0.1073	0.0014	393.5753	0.0467
Nov - Synth	0.1153	0.0004	948.3461	0.0517
Dec - Synth	0.1137	0.0006	435.7339	0.0451
Jan - Synth	0.0961	0.0062	271.7403	0.0280
Feb - Synth	0.125	0.0001	710.1907	0.0290

Table 5.8 Statistical evaluation of dataset 2 on the number of requests per hour against itself to establish baseline values.

	KS Test statistic	p-value	Wasserstein Distance	Jensen Shannon Divergence
Oct - Nov	0.2932	<0.0001	60.1891	0.0757
Nov - Dec	0.3509	<0.0001	61.4647	0.1123
Dec - Jan	0.2644	<0.0001	44.1185	0.0799
Jan - Feb	0.0785	0.0426	13.0929	0.0155

Table 5.9 Statistical evaluation of synthetic data against dataset 2 on the number of requests per hour.

	KS Test statistic	p-value	Wasserstein Distance	Jensen Shannon Divergence
Oct - Synth	0.1378	<0.0001	25.6730	0.0376
Nov - Synth	0.1778	<0.0001	36.1410	0.0269
Dec - Synth	0.1826	<0.0001	27.6826	0.0573
Jan - Synth	0.0865	0.0186	18.0961	0.0130
Feb - Synth	0.0929	0.0090	19.1153	0.0104

cycles, noise at the peaks and troughs, and a slight variation in traffic over a period of seven days.

The results presented in Section 5.2.4 demonstrate that DoWTS generates the appropriate volumes of requests when given a set of input parameters. This is shown by the similarity in scale of value when comparing the real datasets and the synthetic dataset. The mean value of requests in dataset 1 is between 5,131 and 6,245 requests across five months of observed traffic. When given similar usage parameters to this dataset, DoWTS produces 5,415 requests on average. Similar

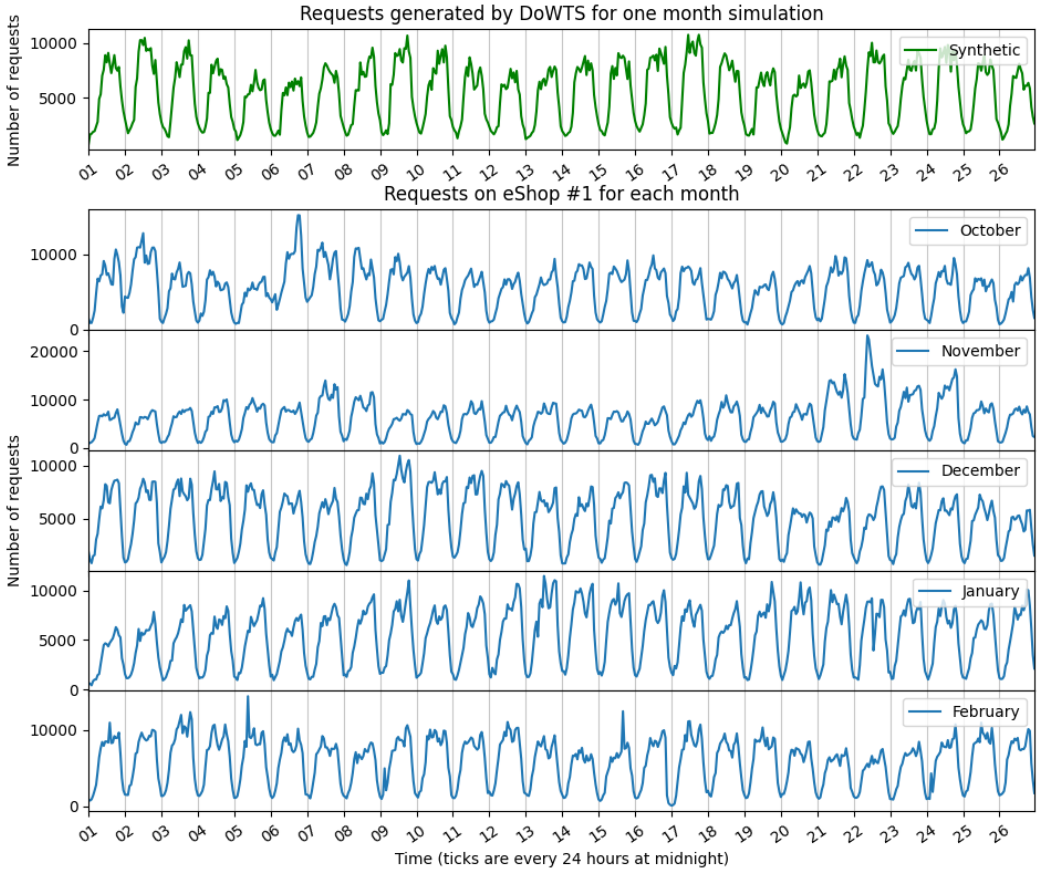

Fig. 5.7 Visual comparison of requests per hour on DoWTS and real dataset 1.

confirmations are produced across the quantitative comparison fields, suggesting the suitability of DoWTS to produce realistic data.

The results presented in Section 5.2.4 should be interpreted as follows.

- **KS Test Statistic -** The KS Test is utilised to determine goodness of fit between two samples. It produces two results, both bounded between 0 and 1: the KS statistic and the p-value. The null hypothesis of the KS Test is that the distributions of datasets are identical if the statistic value is low and the p value is high. In our tests we observe both values to be low. This suggests that we reject the null hypothesis as the distribution of the datasets is not identical. DoWTS does not aim to over-fit the data it produces to the two real datasets utilised in this research, in order to ensure a good variety of produced data. This coupled with the large sample size used in our tests will lead to the KS Test performing overly strictly when calculating the p-value,

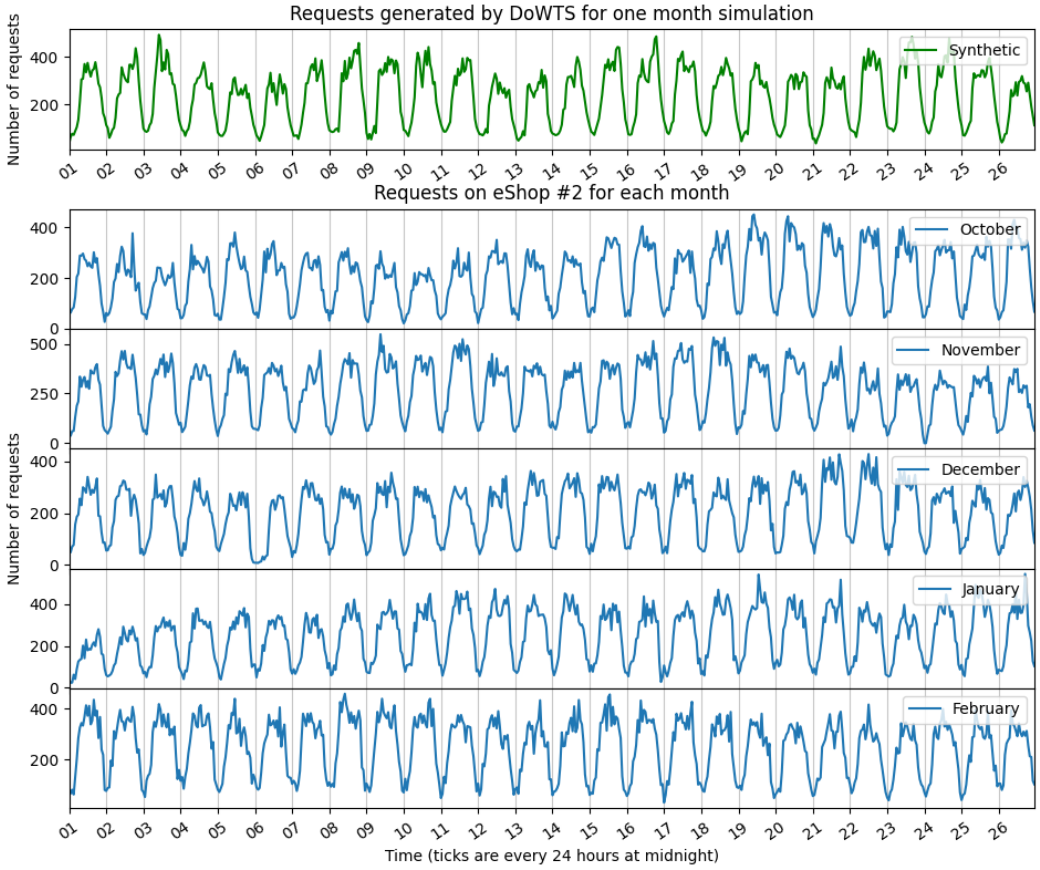

Fig. 5.8 Comparison of requests per hour on DoWTS and real dataset 2.

which results in the recorded low values. However, the calculated statistic of the KS Test (the maximum difference between the distributions) suggests that the recorded values in the datasets are at least similarly distributed, although not exact. The KS Test statistic on each month of real data returns values between 0.09 and 0.19 in dataset 1. DoWTS produces datasets with similar KS Test statistics when compared to each month of real data. This suggests that the data generated by DoWTS is no less believable than any other sample of real data. When coupled with the additional tests, we believe the KS Test statistic to be a valuable marker of the suitability of the data DoWTS produces.

- **Wasserstein Distance -** The value obtained from calculating the Wasserstein Distance has no upper limit. As such, interpretation of the results presented should be independently related to each dataset. The values calculated

when comparing real data are similar to those comparing real with synthetic data.

- **Jensen Shannon Divergence -** The range for Jensen Shannon Divergence is between 0 and 1. The results of our tests demonstrate with suitably low values that DoWTS produces synthetic data similar to the real datasets used for comparison in this research. No comparison of our synthetic data with real data yielded a Jensen Shannon Divergence greater than 0.06.

From our analysis, we can determine that the synthetic usage generator performs suitably well in replicating real traffic. As such, we are confident in utilising it for generating background traffic upon which hypothesised DoW attacks will be launched for use in training of detection and mitigation systems via machine learning.

5.3 Isolation Zone for Denial of Wallet Attack Testing

As DoW is a financial attack, it is impractical to test such on real commercial platforms. Given that there are no known historical examples of DoW, we must have a means of generating data for training of mitigation systems and then validating such systems. To facilitate this, we designed an isolated serverless platform that will emulate the cost damage of DoW and deployment structure of commercial serverless functions. The design consists of a number of machines running on the same network. One machine was designated as the "victim". It would play host to the serverless platform and cost emulator that are being targeted by our attacks. The other machines were designated as "attackers". From these machines, we will launch a variety of attack strategies to determine the best candidate for DoW. The topology of our testbed is illustrated in Figure 5.9.

5.3.1 Configuration

The "victim" machine is a Linux desktop running the Ubuntu 18.04 operating system. The serverless platform used to execute functions is OpenFaaS[2]. OpenFaaS is a framework for building serverless functions with Docker and Kubernetes which

[2]https://www.openfaas.com/

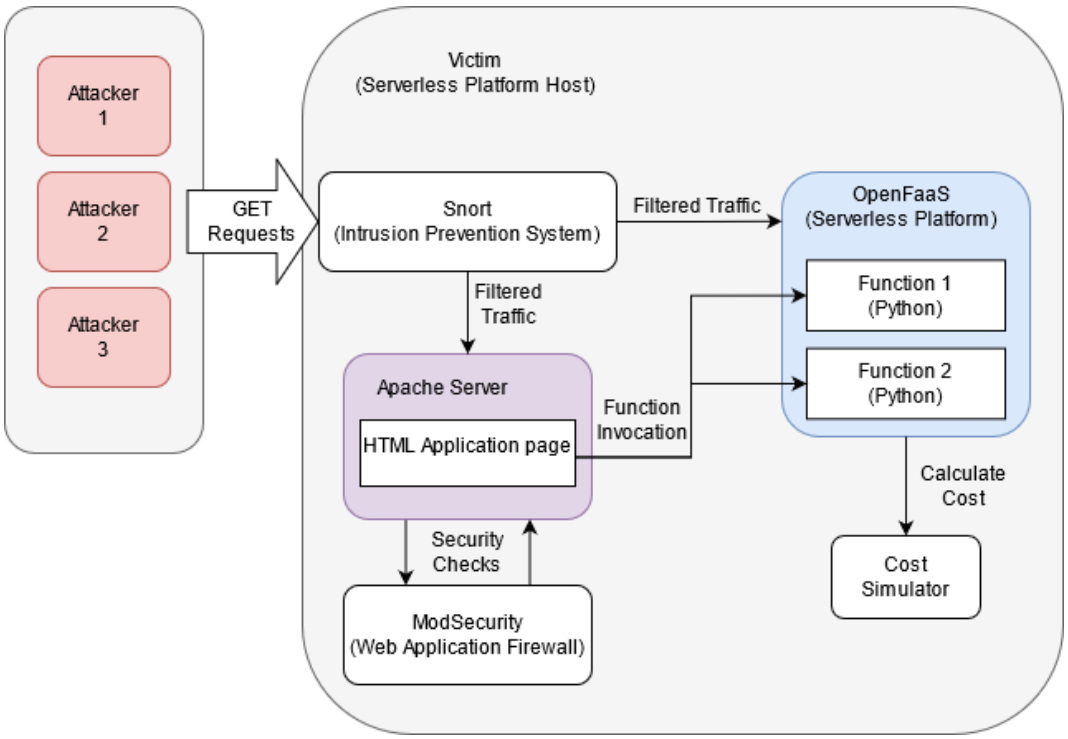

Fig. 5.9 testbed of an isolated serverless platform for attack simulation.

has first-class support for metrics. Any process can be packaged as a function, enabling you to consume a range of web events without repetitive boiler-plate coding. The mock application that will trigger the functions on OpenFaaS runs on Apache Server 2[3]. The Apache HTTP Server Project is an effort to develop and maintain an open-source HTTP server for modern operating systems including UNIX and Windows. In order to perform more meaningful experiments on DoW, our environment has a two-fold security system. Where commercial platforms offer one Web Application Firewall (WAF) that can run on all its products, e.g. AWS WAF[4], finding a single catch-all solution to secure both our application server and serverless platform proved difficult. Our solution was to secure the Apache server with the ModSecurity v2[5] WAF and use the Intrusion Prevention System (IPS) Snort v2[6] to perform WAF actions on the OpenFaaS deployment.

[3]https://httpd.apache.org/
[4]https://aws.amazon.com/waf/
[5]https://modsecurity.org/
[6]https://www.snort.org/

5.3.2 Serverless Deployment

OpenFaaS was chosen as the serverless platform for executing function code. Some of its highlights include;

- Ease of use through UI portal and one-click install
- Write functions in any language for Linux or Windows and package in Docker/OCI image format
- Portable - runs on existing hardware or public/private cloud with Kubernetes or containerd
- CLI available with YAML format for templating and defining functions
- Auto-scales as demand increases

OpenFaaS runs on the "PLONK" stack. PLONK is a Cloud Native stack for building applications which stands for:

- Prometheus - metrics and time-series
- Linux - OS or service mesh
- OpenFaaS - management and auto-scaling of compute - PaaS/FaaS, a developer-friendly abstraction on top of Kubernetes. Each function or microservice is built as an immutable Docker container or OCI-format image.
- NATS - asynchronous message bus / queue
- Kubernetes - declarative, extensible, scale-out, self-healing clustering

The function gateway can be accessed through its REST API, via the CLI or through the UI. All services or functions get a default route exposed, but custom domains can also be used for each endpoint. Prometheus collects metrics which are available via the function gateway's API and which are used for auto-scaling. Functions are invoked from the application front end via an API gateway. This also exposes the functions to direct invocation (as is normal in FaaS). The front-end and OpenFaaS communicate via HTTP requests and JSON.

5.3.3 Web Application Server Deployment

The Apache server hosts the application front end. It is a powerful, flexible, HTTP/1.1 compliant web server that implements the latest protocols, including HTTP/1.1 (RFC2616). It is highly configurable and extensible with third-party modules, which we make use of when implementing our security solution. Apache runs on Windows 2000, Netware 5.x and above, OS/2, and most versions of Unix, as well as several other operating systems, meaning that we could easily pair it with our OpenFaaS deployment on Ubuntu 18.04. This server will be used to host mock applications for future experiments.

5.3.4 Security

As mentioned above, where commercial platforms offer a single WAF that protects all their products, this proved unrealistic for our environment. We split the security measures into 1. protect the application server and 2. protect the serverless platform.

The application server is protected by ModSecurity. It is a toolkit for real-time web application monitoring, logging, and access control. Its capabilities include:

- Real-time application security monitoring and access control
- Full HTTP traffic logging
- Continuous passive security assessment
- Web application hardening

We use this a the primary means of securing the application from attacks such as XML or Shell injection etc.

A highly recommended means of mitigating HTTP Flood attacks (which is the core of worst-case scenario DoW attacks) is rate limiting based on IP address. For this, all inbound traffic gets filtered through Snort. Snort is an open-source network intrusion detection system (NIDS). Snort is a packet sniffer that monitors network traffic in real time, scrutinising each packet closely to detect a dangerous payload or suspicious anomalies. Snort is based on libpcap (for library packet capture), a tool that is widely used in TCP/IP traffic sniffers and analysers. Through

protocol analysis and content searching and matching, Snort detects attack methods, including DoS, buffer overflow, CGI attacks, stealth port scans, and SMB probes. Running in "inline" mode, Snort becomes an intrusion prevention system. Inline mode is where Snort sits between two bridged network interfaces, analysing network packets as they come in. When suspicious behaviour is detected, it will drop any packets that trigger certain rules. Otherwise, it will pass them to the Apache server or OpenFaaS back end (depending on the function invocation method).

5.3.5 Serverless Platform Pricing Emulator

We developed the Serverless Platform Pricing Emulator in order to compare the effects of DoW across multiple commercial platforms. It operates largely the same as the DoWTS Serverless Platform Emulator. However, it is designed to sit on top of an OpenFaaS serverless deployment that allows you to emulate how much its function executions would cost on AWS Lambda, Google Cloud Functions, IBM Cloud Functions, and Microsoft Azure Functions. It is written in Python and converts usage metrics gathered by Prometheus on the OpenFaaS platform to a comparable cost had those functions been invoked on the four largest commercial serverless platforms. The cost calculator queries Prometheus for the total number of successful function invocations and the cumulative execution time. As all functions are configured to 128MB, delineation between function executions is not required and a total cost can be calculated as per each platform's pricing guidelines. The source code[7] is available for use by the community who wishes to perform similar experiments on serverless computing financial effects.

5.3.6 Attacking Nodes

The attack machines execute a Python script that sends multiple GET requests to the functions and changes the IP address of the machine (simulating any number of attack nodes required). For DoW, slow rate attacks are the primary concern, however, DoS style of attacks may also be executed.

[7]https://github.com/psykodan/openfaas-commercial-platform-emulator

The attacker can also be configured with the Kali Linux operating system[8] for the use of existing attack software. In Section 5.3.7, we do so and utilise the DoS tool Low Orbit Ion Cannon.

5.3.7 Demonstrating Denial of Wallet

Using our testbed, we can demonstrate the mechanism of DoW attacks. First, we use an existing DoS tool, LOIC, a DoS tool made famous by the "Operation Payback" attacks in defence of Wikileaks in 2010, which can perform HTTP flooding attacks [32]. We configured a single node with Kali Linux and ran LOIC with a single serverless function on our OpenFaas platform as the target. We allowed LOIC to execute for one hour, continuously spamming direct function invocations to our OpenFaaS deployment. Our platform gathers metrics such as function invocation rate, container scaling, total invocations, and execution time using Prometheus and Grafana. We observed that more than 130,000 function invocations could be triggered in this time span (Figure 5.10). The average invocation rate and run time per 20 second interval is also recorded (Figures 5.11 and 5.12). This run demonstrated how a serverless platform scales to accommodate large volumes of requests by creating additional containers for function execution (Figure 5.13). However, this example is a worst-case scenario where a single attacker is allowed to continuously spam requests. A WAF can mitigate simple GET request flooding using rate limiting. Using LOIC again and activating our intrusion prevention system (IPS) that acts as a WAF to our OpenFaaS deployment, we can limit the damage any one node can do with an HTTP Flooding attack by rate-limiting requests from a single source. This would suggest that short time span single-node attacks are not suitable for DoW.

From this demonstration, we can infer that an unprotected serverless function is susceptible to DoW from a single node. Implementing rate limiting protects against the issue. However, as discussed previously in Section 3.3.1, DoW can be executed over a much longer time span than a DoS attack, thus rate limiting will be ineffective in such cases.

[8] https://www.kali.org/

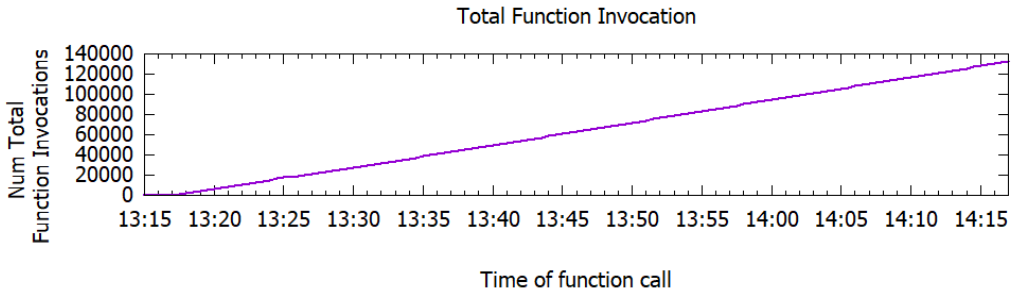

Fig. 5.10 Total function invocations collected from HTTP flood attack without protection.

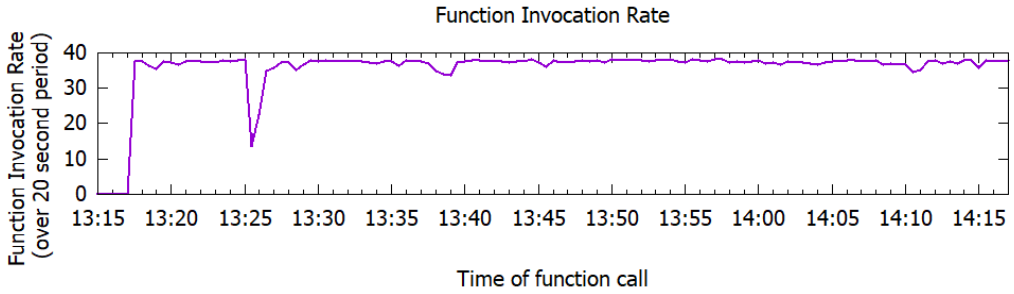

Fig. 5.11 Function invocation rate per 20 second interval collected from HTTP flood attack with no protection.

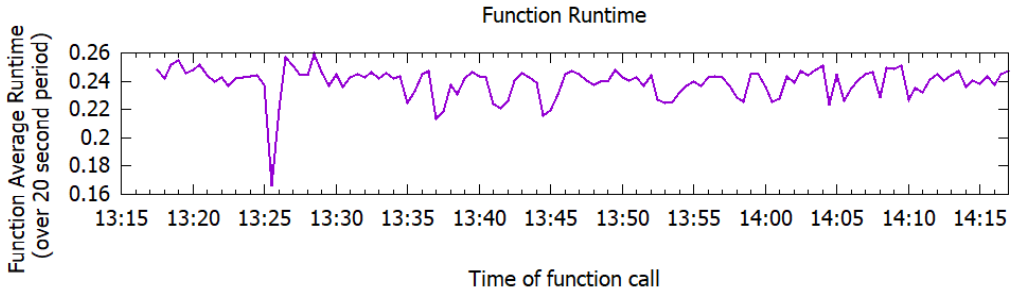

Fig. 5.12 Average run time of the function per 20 second interval collected from the HTTP flood attack with no protection.

5.4 Summary

In summary, this chapter has detailed the creation of two tools that allow for the continued research on DoW attacks without the need to use commercial platforms that would end up costing the researcher unnecessary amounts. One

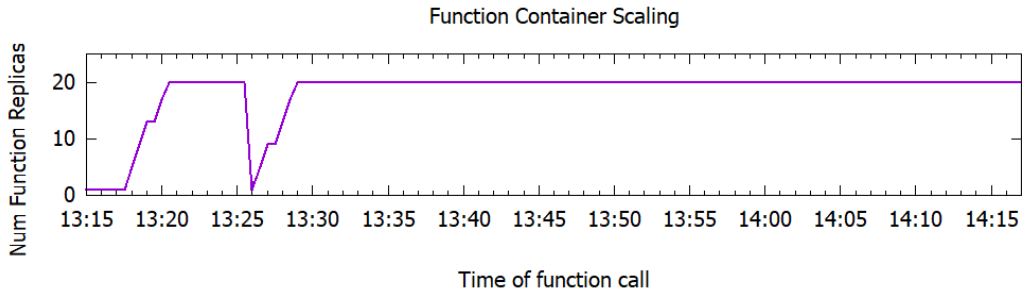

Fig. 5.13 Metrics collected from HTTP flood attack with no protection. Function replica generation

that can generate large quantities of normal usage data on theoretical serverless applications. The other that allows for the deployment of serverless functions on top of a system that calculates equivalent bill amounts on four of the largest commercial serverless platforms. Together, these tools can be used to rapidly test scenarios of attack and defence strategies.

This work addresses our second research question, in that we were able to overcome the limitation of a lack of historical data of usage on serverless platforms and of the attack itself. This also confirms our hypothesis that synthetic data can be utilised as a source data for training of mitigation systems (as will be discussed in the next chapter).

This chapter highlights the difficulties that had to be overcome in this project. The solutions proposed are the first step in useful tooling for DoW research. As such, we believe that this work serves as a significant contribution to the field, whether it is via use of our tools or as inspiration for further refined solutions.

CHAPTER 6

A Novel Solution to a Unique Problem - Detecting Denial of Wallet

This chapter presents the culmination of the previous work, taking the looming threat of DoW and proposing a means of mitigating this attack before it becomes a wide-spread issue. Our system of DoW detection utilises a novel means of representing large-scale traffic logs in a graphical form that consumes a fraction of the memory it would in its raw form. Based on these graphical representations, a CNN is trained to detect anomalous patterns in traffic. This detection system may serve as an early warning system for application owners to take action when there is suspected abuse of serverless functions leading to DoW. This chapter contains work currently submitted to the Oxford University Press Journal of Cybersecurity for review. This chapter answers *RQ3* "Can an attack that so closely mimics regular traffic be detected and mitigated against?".

6.1 Introduction

Recent research on DoW has further defined it be split into *internal* and *external* DoW [138]. This research suggests that external DoW can be detected and mitigated by traditional DoS mitigation techniques. However, unlike DoS, DoW

may use alternative attack patterns that do not trigger DoS detection parameters as they may not be volumetric attacks, generating large traffic volumes that seek to cause damage in a short amount of time. Instead, these attacks execute over long periods of time in order to not raise suspicion like a traditional DoS flooding attack (leech attacks).

In this chapter, we present our novel approach to representing these attacks for use in a detection system that utilises deep learning of a CNN for classification of suspicious traffic on an application. The contributions of this body of work are as follows:

1. Development of a novel approach to translating large volumes of request traffic data into summary image representations and then generating a new dataset of the said representations via data synthesis.

2. Analysis of the feasibility of approaching DoW detection as an image classification problem and subsequent training of a CNN for malicious traffic classification.

3. Creation of a system for applying the traffic classification model that would allow application owners to evaluate their traffic in real time and alert them when potential attacks have been detected.

6.2 Denial-of-Wallet Attack Data Generation

As the aim of DoW is to cause the application owner to receive excessive usage bills, the users of the application are not the ones being affected by the attack. Therefore, if there was to be an attack, it would not serve the victim any advantage to publicly disclose such an attack. This, along with the relative recency of the development and uptake of serverless computing, is potentially why there are no documented large-scale DoW attacks that can be used as training data. We utilised DoWTS [129] in order to generate many different attack scenarios with varying parameters for attack configuration.

6.2.1 Image Representation of Function Request Traffic

The logs generated by DoWTS cover one month of traffic, which is the billing cycle of a serverless application. These logs can vary in size depending on the normal background usage and the intensity of the attack. For the normal traffic use case used by default in DoWTS, these files can expect to be approximately 4GB in size with no attack execution on top. The subsequent attacks can increase this to 20GB in cases where the attack is particularly aggressive (high rate of requests). As such, a method of representing the data to minimise storage space and computation time to process is required.

The traffic data is represented as a heat map, with hour of the day on the x axis, day of the month on the y axis, and number of requests relative to the month's worth of requests as the intensity of the square on the grid (as demonstrated in Figure 6.1). These heat maps are 24×30 pixels in size and single channel greyscale images in order to reduce storage and processing time. The intensity of the traffic is represented as a value of 0 to 255 relative to the traffic in that month. This procedure is outlined in Figure 6.2. The actual heat maps produced for the three types of attack modelled by DoWTS are shown in Figure 6.3. Their pixels are not as clearly visible because of scaling up from their original small size.

6.2.2 Dataset Characteristics

Using DoWTS to run various attack scenarios and then converting those logs into heat maps, we generated the following dataset:

- **10,000 images generated in total**
  - **4,000 Normal traffic heat maps** - Traffic modelled by DoWTS based on historic web application use data
  - **3 attack types** - Randomly choosing a start time as the hour in the month and the duration of the attack in hours
    - **2,000 Linear attack heat maps** - Randomly chosen from range, fixed rph for each run. Range between 0 and 3000 rph.
    - **2,000 Geometric attack heat maps** - Variable fixed request increase factor. Range between 1.001 and 1.01 increase factor per hour.

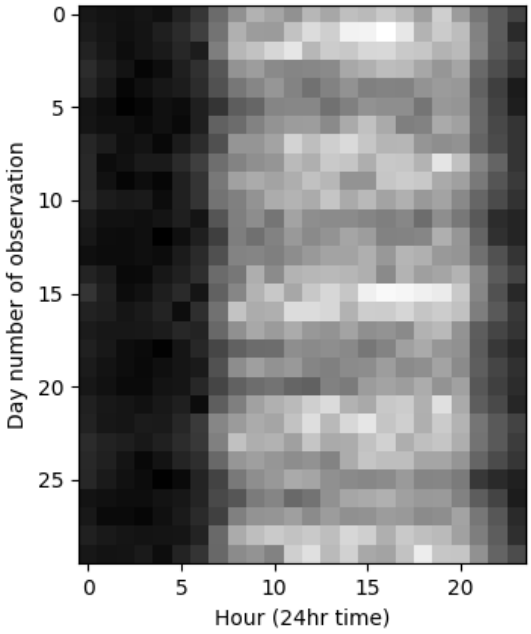

Fig. 6.1 Representation image of a produced heat map with clearly visible pixels where (x, y) co-ordinate of pixel corresponds to (hour, day) of observation.

* **2,000 Random attack heat maps** - Variable rph each hour. Range between 0 and 3000 rph.

This dataset is fully labelled for the four classes; normal, linear, geo, and rand. There is also an accompanying log file that lists the attack parameters and the damage caused. However, it is not used for our detection system.

In summary, our dataset consists of 10,000 labelled images of heat maps. These heat maps are labelled for the traffic they represent on a serverless application, i.e., the number of function invocations an hour. The traffic representations are: normal use traffic, linear increase attack traffic, geometric increase attack traffic, and random rate increase attack traffic. The data produced cover a range of attacks intensity for training of a robust detection model.

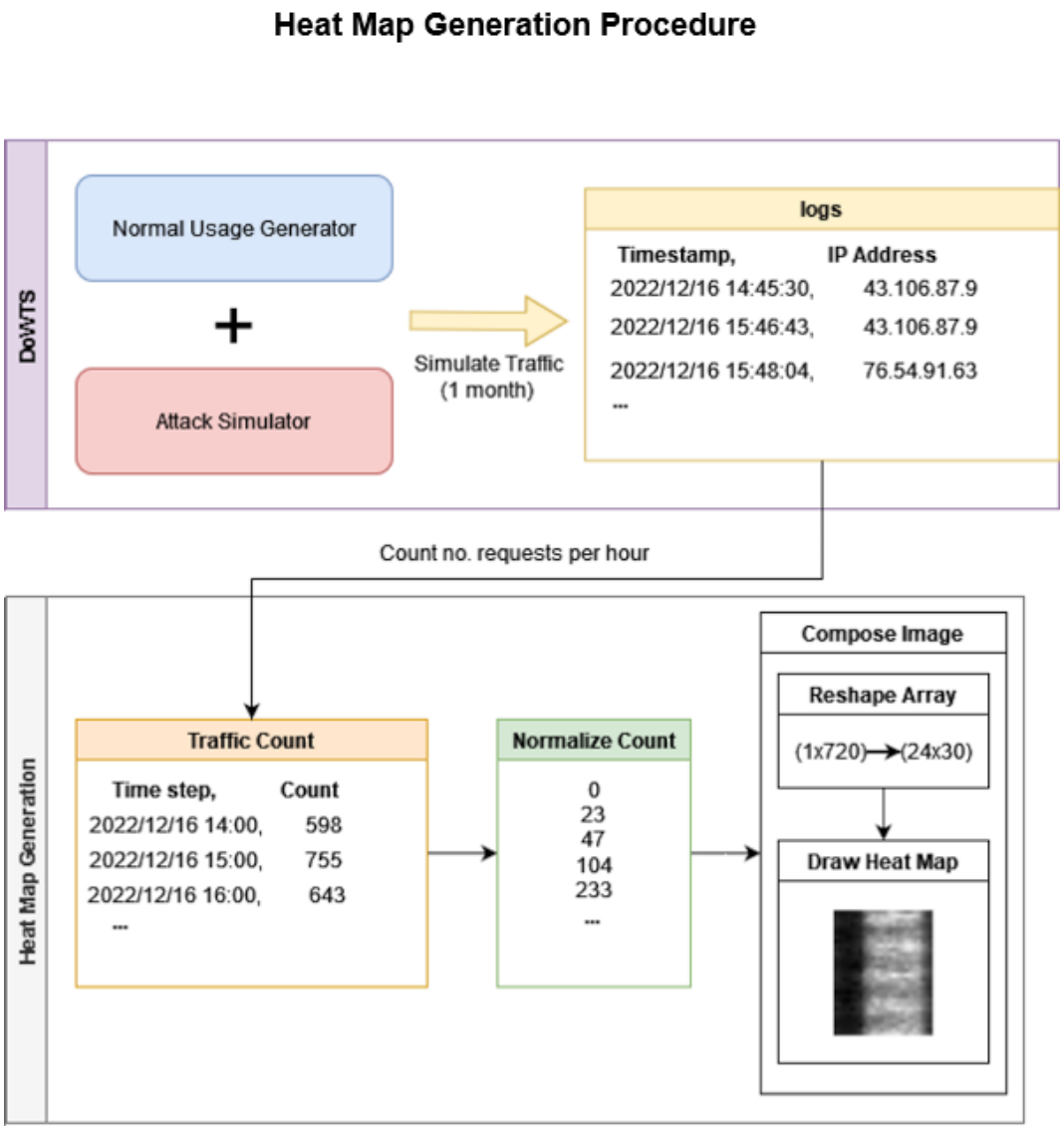

Fig. 6.2 Full life-cycle of data generation from DoWTS to heat map creation.

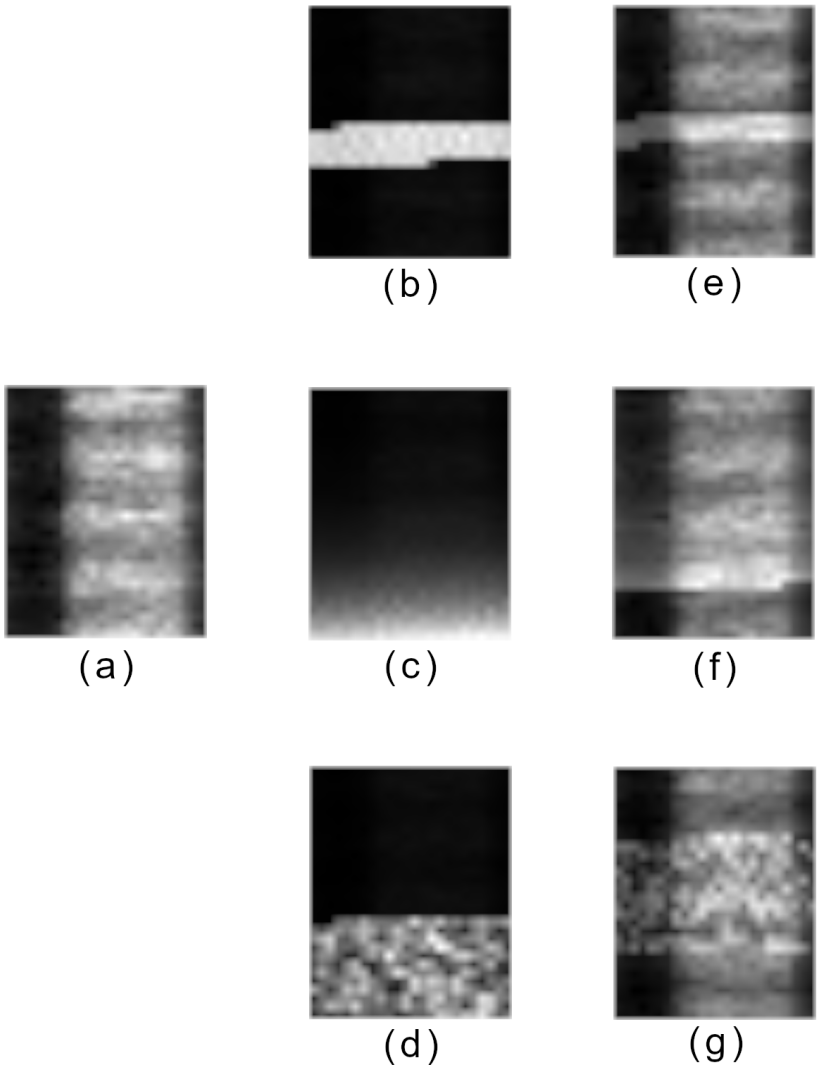

Fig. 6.3 Heat map representation of request traffic: (a) Normal Traffic, (b) Linear rate attack, (c) Geometric rate attack, (d) Random rate attack, (e) Difficult to detect linear rate attack due to lower attack intensity, (f) Difficult to detect geometric rate attack due to lower attack intensity, and (g) Difficult to detect random rate attack due to lower attack intensity. Note that the images are scaled up from 24×30 pixels for visibility

6.3 Image Classification Model for Detection of Attacks

6.3.1 Analysis of Preexisting Networks for Image Classification

In Section 2.4.1, we outline a number of image classification CNNs that achieved notable results on the ImageNet dataset. These networks are publicly available as Keras Applications [139]. As such, we perform an evaluation of a the selection of networks, discussed in Chapter 2, on our dataset for the DoW attack detection problem domain. Table 6.1 lists the classification accuracy, F1 score, precision and recall of each model as defined in equations 6.1, 6.2, 6.3 and 6.4 respectively, where $TP = TruePositives$, $TN = TrueNegatives$, $FP = FalsePositives$ and $FN = FalseNegatives$. Also included are a basic implementation of a Multi Layer Perceptron (MLP) and a Long Short-Term Memory (LSTM) network.

All models were trained on 1500 images of normal, linear attack, geometric attack, and random attack traffic each. The images were scaled up by a factor of three in order to meet the minimum input size for the models. They were also loaded as three-channel images. An 80/20 training to validation split was used on this data, where the validation set is used for tuning of hyperparameters in the model. A final dense layer was added to each network as the output layer for classification of the four labels using a *softmax* activation. Each model was trained for 10 epochs to ensure a reasonable comparison. A holdout dataset was used for evaluation containing 2000 images of even ratio between the four classes with 500 examples from each class. The model is then trained again on an 80/20 split with the samples shuffled a further 9 times. This process is visualised in Figure 6.4.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{6.1}$$

$$F1 = \frac{2 \times TP}{2 \times TP + FP + FN} \tag{6.2}$$

$$Precision = \frac{TP}{TP + FP} \tag{6.3}$$

$$Recall = \frac{TP}{TP + FN} \tag{6.4}$$

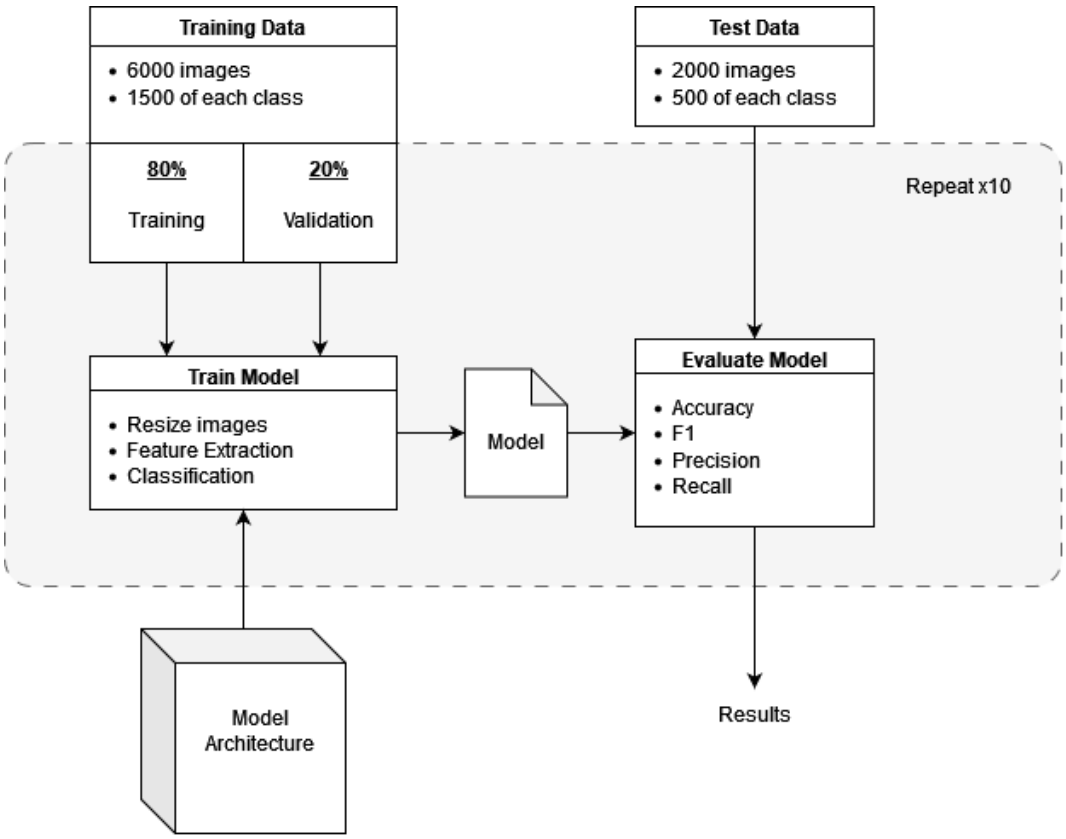

Fig. 6.4 System of training and evaluation of models

Table 6.1 Comparison of preexisting image classification models and two non-CNN based models

Model	Accuracy (%)	*F*1 (%)	Precision (%)	Recall (%)
MobileNetV2[70]	95.66	95.79	95.85	95.66
ResNet50[65]	98.03	97.29	98.13	97.93
SqueezeNet[66]	**98.44**	97.12	**98.49**	**98.44**
VGG16[64]	97.48	96.92	97.62	97.43
Xception[68]	98.34	**97.78**	98.43	98.24
MLP	73.76	62.04	74.44	72.60
LSTM	91.98	81.92	92.21	91.32

From the results shown in Table 6.1, all five CNNs achieve high results across the four metrics. The LSTM achieves good accuracy; however, the lower F1 score suggests a higher number of false classifications. The MLP performs significantly worse than the other models. We believe the further exemplifies how effective CNNs are compared to these neural networks. Any of the five pre-designed net-

works would make good candidates for implementation in a deployable detection system, given the high results demonstrated. However, we observe that the network with a much more simplistic architecture, VGG16, performs almost as well as three of the higher-complexity networks and better than MobileNetV2.

6.3.2 Creation of a Domain Specific Model

As the problem domain of detecting DoW attacks, at present, is limited to the three proposed basic attack patterns [129], it is unnecessary to employ the networks discussed above. We deduce that the use of one of these networks designed to classify 1000 classes (from the ImageNet dataset) is excessive. As such, in this section, we outline the design and performance of a model of significantly lower complexity.

We shall henceforth refer to our proposed model as *Denial-of-Wallet Network (DoWNet)*. The network architecture is shown in Figure 6.5. Inspired by the architecture of VGG16, DoWNet comprises 3 convolution layers and 4 dense layers. Pooling is performed after each convolution layer, and a dropout layer is included before the dense layers to prevent overfitting. Each convolution layer uses Rectified Linear Unit (ReLU) activation and filters of size 3. The output layer uses softmax activation for multiclass classification. Images are loaded in their original 24×30 size and as a single channel image. This further reduces complexity by not requiring the search for cross-channel correlation.

The results shown in Table 6.2 demonstrate that DoWNet performs almost as well as the top three classification models in Section 6.3.1, and outperforms VGG16 and MobileNetV2. We do not claim that DoWNet is a better performing model than these models. However, since this classification task is not as difficult as the ILSVRC, we have shown that a more simple model can achieve similar performance. The advantage of this is that there is less time required to train and use the model. SqueezeNet was the model that performed best in our tests with a classification accuracy of 98.44%. It took 777.79 seconds to train the model. Whereas, DoWNet with an accuracy of 97.68% took 11.8 seconds. There is great importance in these short training times, as this attack is so new, it is constantly evolving, and retraining will be continuous beyond the attack patterns demonstrated in this chapter. The size of the files created when saving the models for use in further applications was also notably different with SqueezeNet generating

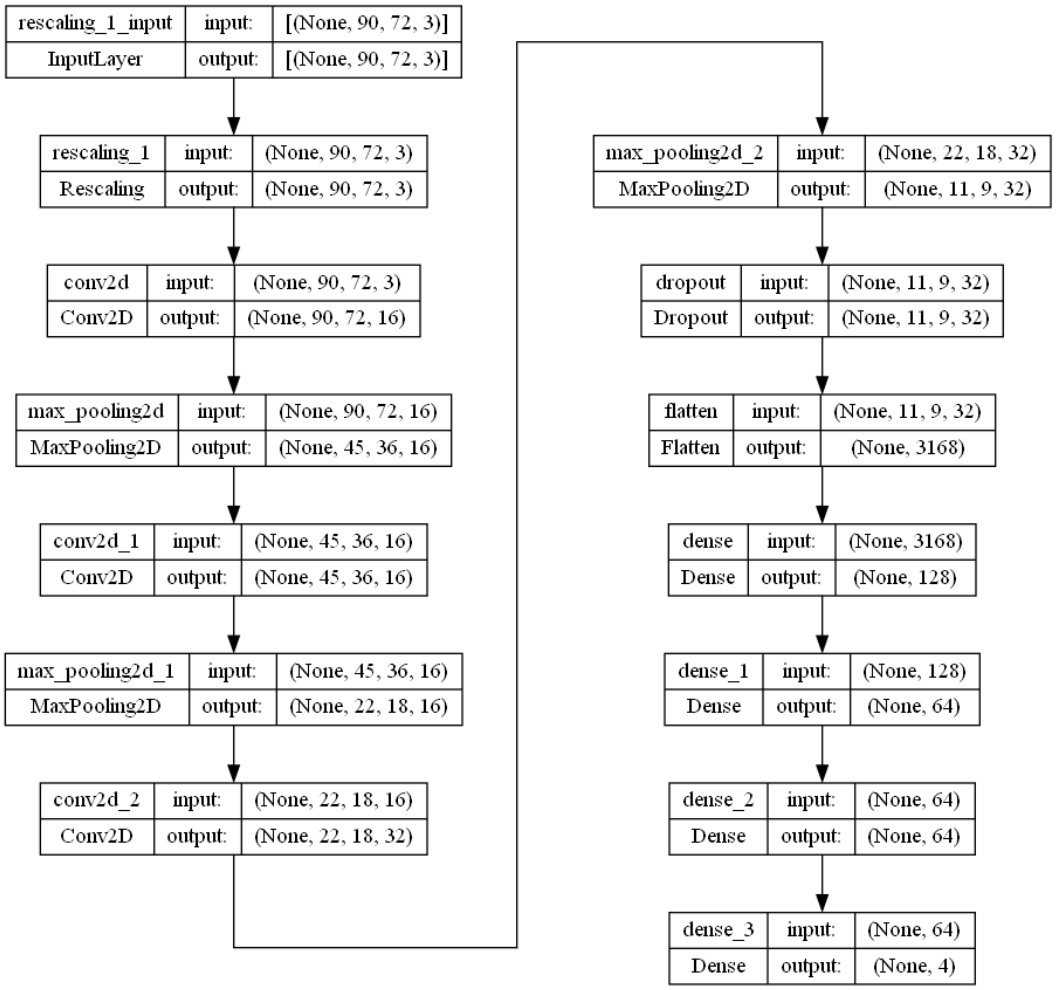

Fig. 6.5 DoWNet layer architecture

a 1081 KB file and DoWNet generating a 231 KB file. The reduced size and time overhead make our simple model more suitable for use in a lightweight detection system that would sit on top of a serverless application. Figure 6.6 demonstrates DoWNet's classification on the holdout test dataset.

Where there were misclassifications, they were predominantly heat maps that had such little attack traffic that it looks like a normal traffic heat map. These low attack rate heat maps were generated as a result of the range of rph for attacks starting at 0. We believe these examples are important as they make the model more robust and capable of detecting attacks that are not just incredibly high rate (that emulate DoS flooding attacks). We relate this to cat vs. dog classification where there may be images of dogs that look like cats and visa versa.

Table 6.2 DoWNet performance metrics

Model	Accuracy (%)	F1 (%)	Precision (%)	Recall (%)
DoWNet	97.98	97.92	97.98	97.98

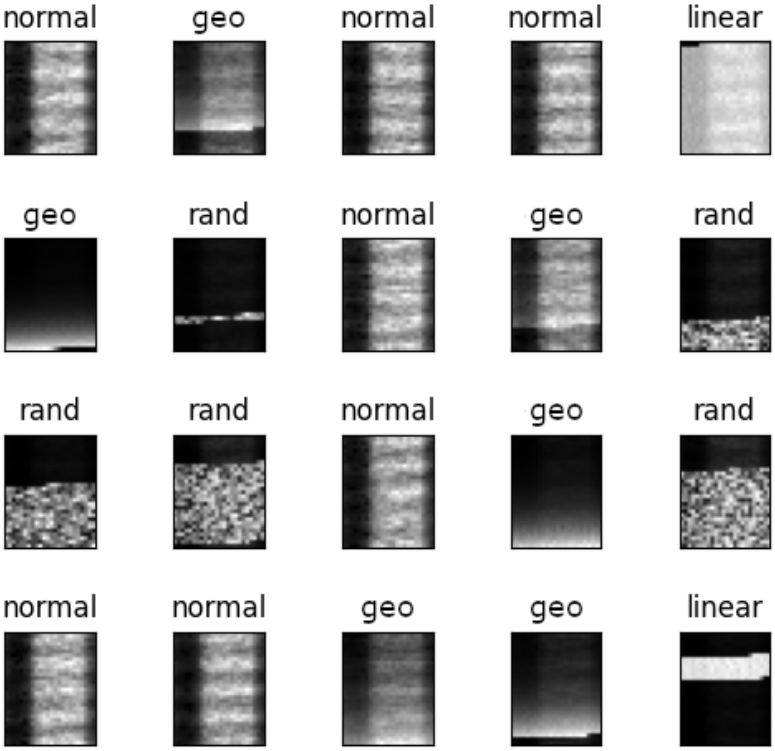

Fig. 6.6 DoWNet successful classification of test data

6.4 Implementation of Model in Deployed Application Scenario

DoWNet is capable of accurate classification on heat maps that represent one month's worth of request traffic. As such, this approach is limited to checking if an application is under attack once a month, as you would need to have gathered the request log data to compose a new heat map. In order to address this limitation, we propose a method of utilising DoWNet that allows for daily assessment of request traffic for signs of attack.

6.4.1 Pixel Streaming Heat Map

As DoWNet is trained on images of a month's worth of request traffic represented as a heat map, it must be supplied with a full image to perform classification. Our solution is to continually update the image pending new values from the recorded traffic of the previous hour. We term this *pixel streaming* as our images are 24×30 pixels images where each pixel represents an hour's traffic intensity at a specific time during the month. By continually streaming in new values for pixels in a sliding-window fashion, we can update the image to represent the traffic trends in the heat map. There are two key aspects that allow this to work:

1. There must be a baseline month of data for the first use. A heap map must be generated for the updates to take place on. Upon this heat map, the streaming will commence and from then will continue to use the previously streamed data as the historical traffic values.

2. All values for each pixel in the image are re-normalised with each timestep. The actual count of rph that dictate the image's pixels is separate from the normalised values represented in the heat map. As new requests come in on each hour, those values are updated, and a new normalised array is generated for composition of the heat map.

Figure 6.7 demonstrates how pixel streaming and DoWNet are used together for DoW classification. The top-left heat map is determined to be normal traffic. In this example, this happens to be the initial baseline traffic required for the pixel streaming. However, it could also be the product of months of streaming where there were no attacks taking place. The top-right heap map is the result of 10 days of streaming. As each pixel represents an hour, there have been 243 iterations of updates and normalisation to the heat map. Visually, we can see that a faint line is developing and DoWNet is now classifying the traffic as a random increase rate attack. In this example, an geometric increase attack is being launched at this time. The bottom-left heat map shows further updates to the heat map after 18 days. DoWNet had previously been classifying the attack as random and then linear increase rate until the features present were enough to be classified as geometric. The final heat map is after the full run of one month with the geometric attack.

This solution serves as an early warning system that an attack is taking place. At the 10 day mark where DoWNet first became suspicious of the traffic, an ap-

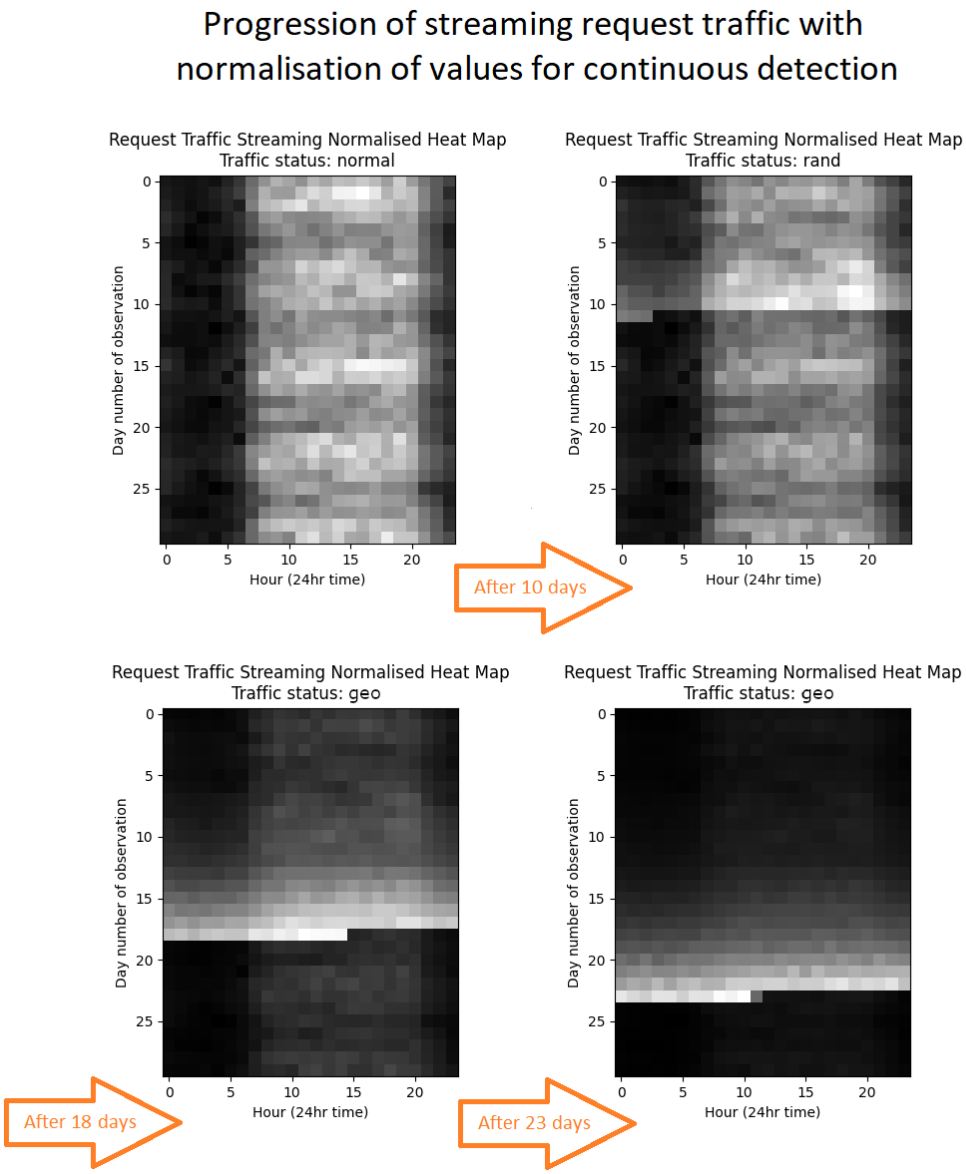

Fig. 6.7 Progression of streaming the total number of requests for each hour and normalising the data to create a new heat map for analysis. The scenario shown is an geometric attack that starts roughly on day 5 and increases the rate of requests for 20 days. The detection model initially classifies the attack traffic as random until more data is available before it is correctly identified as an geometric attack.

plication owner can take appropriate measures to investigate and secure their endpoints before the full damage of the attack can occur, such as the mitigation approaches discussed in Chapter 3.

6.4.2 Application of Solution on Deployed Service

We believe that an appropriate deployment of DoWNet should have as little impact on the architecture of an application as possible, since it should only run once an hour. As such, it is a prime candidate for being written as a serverless function itself. Deploying DoWNet into a function that lives within the same space as the rest of the application means that there is no requirement to parse data to a third party, thus mitigating potential breaches in privacy. Another great advantage is that the whole system runs on the developers cloud platform, allowing for ease of maintenance and setup.

We present a proof-of-concept system that operates on the OpenFaaS serverless platform. Our testbed follows the isolation zone for serverless attack testing outlined in Chapter 5. An additional MongoDB database was used for persistent storage of the traffic count at each hour.

DoWNet was written into a function that executes Python code. The function was written in such a way that it could be translated to other serverless platforms, remaining as *platform agnostic* as possible. The operation of the function is as follows:

Prerequisites

1. NoSQL database. Each major platform offers its own solution. Our example uses MongoDB.

2. Data collection of 720 hours ($24 hours \times 30 days$) of traffic counts. This is required to generate the first heat map.

3. Document that contains the current hour number and previous total invocations.

Function

1. Connect to the database and get the current hour and the previous total for the new value overwrite.

2. Utilising the platform's provided metrics system, fetch the total count of function invocations. OpenFaaS uses Prometheus, whereas AWS uses its own product, CloudWatch. These metrics are available via API requests.

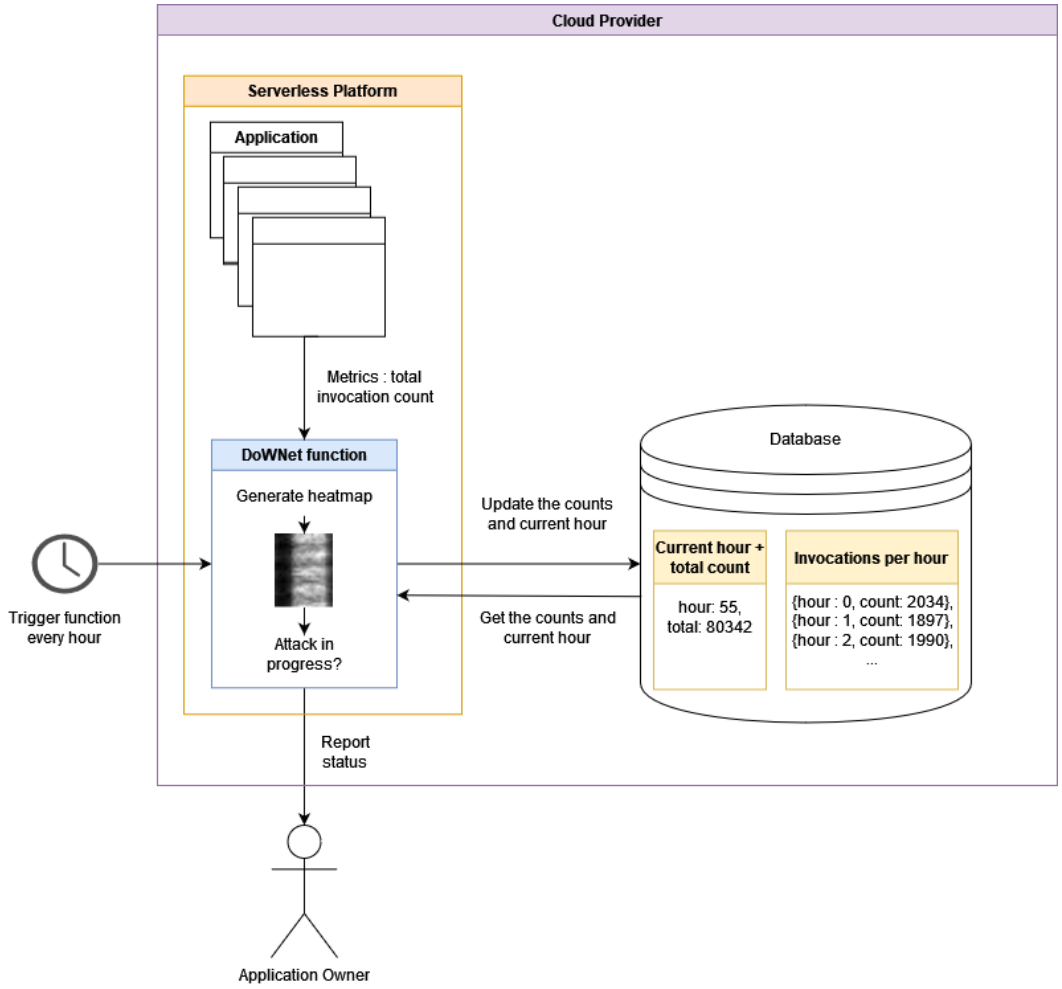

Fig. 6.8 Operation of proof-of-concept deployment of DoWNet.

3. Calculate the total for that hour using the previous total. Update hour invocation count, new total and new current hour in database.

4. Shape the data with the new hour count into the 24×30 grid and normalise into heat map.

5. Apply the DoWNet model to heat map.

6. Report results.

The operation of this system is shown in Figure 6.8

6.4.3 Results

An experimental testbed was set up that consisted of:

- A dummy application to represent an application
- The deployment of DoWTS
- A traffic generator utilising the same synthetic traffic generator as in DoWTS
- A pre-loaded collection of a months traffic with the usage set to not exceed 2500 requests per hour. This is a low-scale example to avoid the need to use a botnet for sending attack requests. As DoWNet is trained on normalised data, the reduced scale should not affect the results.

Two control tests were run to establish that the traffic generator when executing only normal traffic does not trigger false positive detections. The system ran for 24 hours with the synthesised traffic starting on hour 48 (in order to emulate a system that has already been running for a period of time). Figure 6.9 shows the change in the heat maps over 24 hours in both tests. Figure 6.10 shows the prediction values for each traffic classification over the 24-hour period.

An attack was then launched on the dummy application, starting on hour 115 of the month, and used the random rate increase attack vector. The attack was detected successfully after 5 hours. Figure 6.11 demonstrates the effect of the attack on the generated heat map and Figure 6.12 graphs the prediction values of DoWNet.

The results show a clear success in detecting DoW attacks. The function running DoWNet executes in under 2 seconds, thus requiring little to no cost over head to run given its low-frequency invocation (once an hour).

6.5 Discussion

6.5.1 Challenges of Denial-of-Wallet Detection

As discussed in this chapter and in previous literature [83, 129], *external* DoW attacks need not simply be DoS style flooding attacks. The threat of lower intensity leeches would not be detectable by traditional DoS mitigation's such as rate

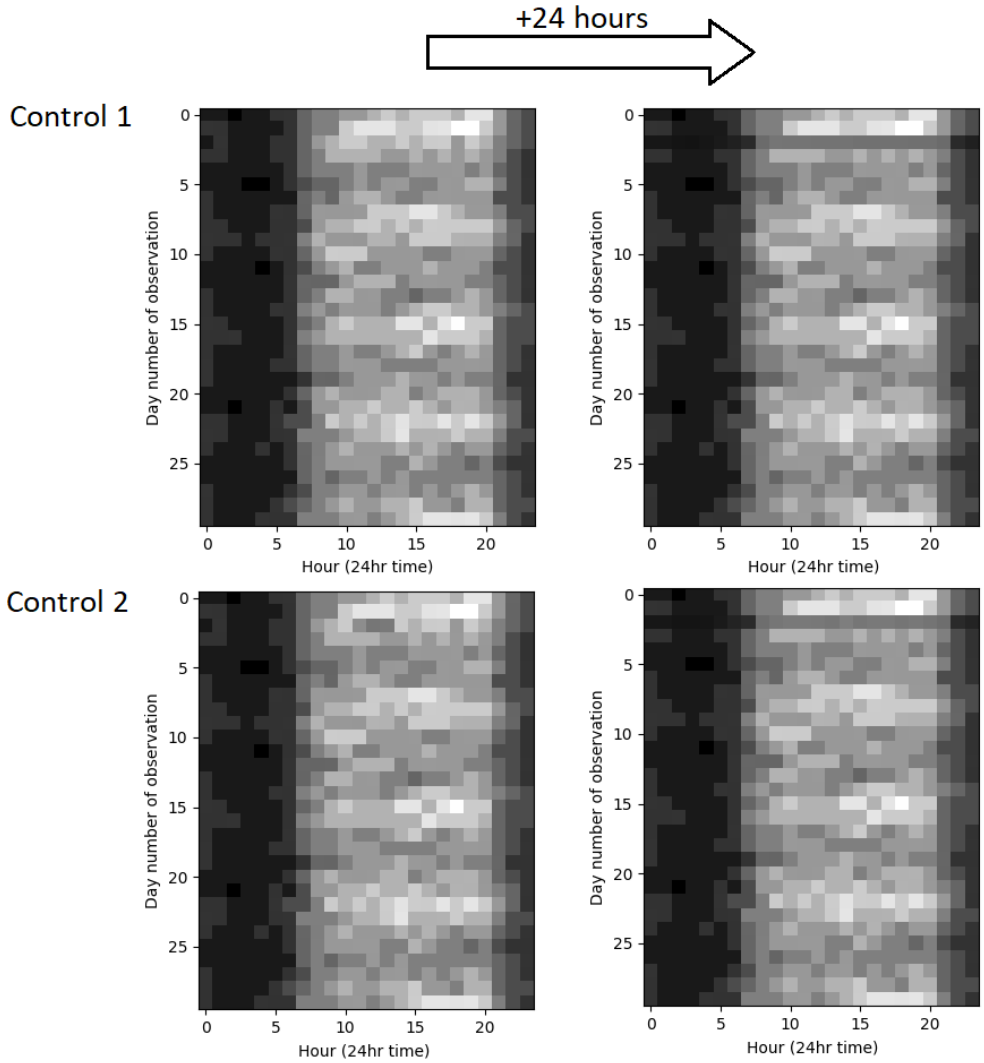

Fig. 6.9 Change in the heat maps over 24 hours in control runs

limiting or packet analysis as individually they would be indistinguishable from normal traffic. Therein lies the initial challenge with DoW detection that we have set out to solve.

Through the translation of traffic into small heat map images, we were able to effectively represent long spans of data to facilitate training our CNN. As with leech attacks, they are most effective over these long time spans. The heat map representation is also robust in that since the values are normalised, DoWNet is trained to recognise patterns rather than suspicious values. This would be the

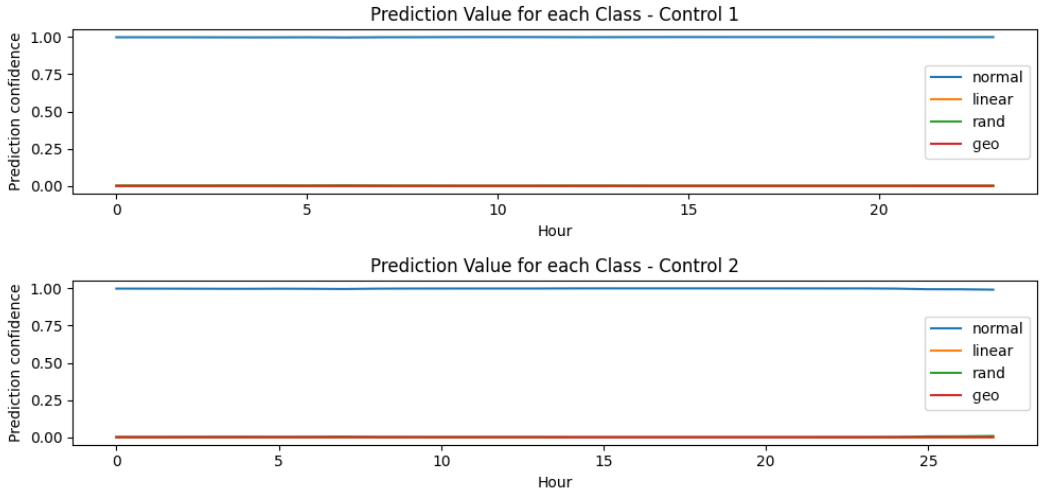

Fig. 6.10 Prediction value of each class over 24 hours in control runs

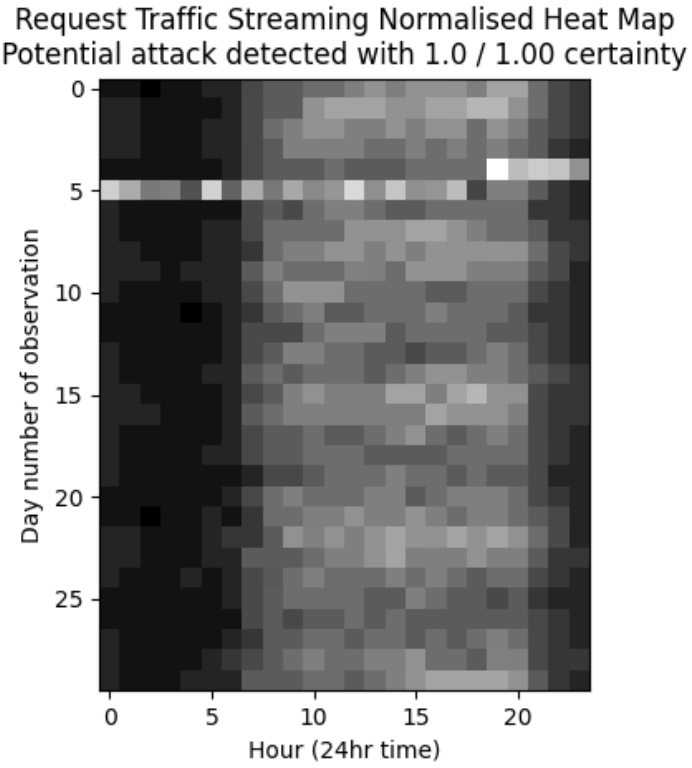

Fig. 6.11 Resulting heat map after the 24 hour attack scenario. Note: 1.0/1.00 is how Tensorflow represents 100% when displaying results from classification.

case if we treated the requests individually. Reoccurring IP addresses could be easily detected via ML using a clustering algorithm. However, this would not be

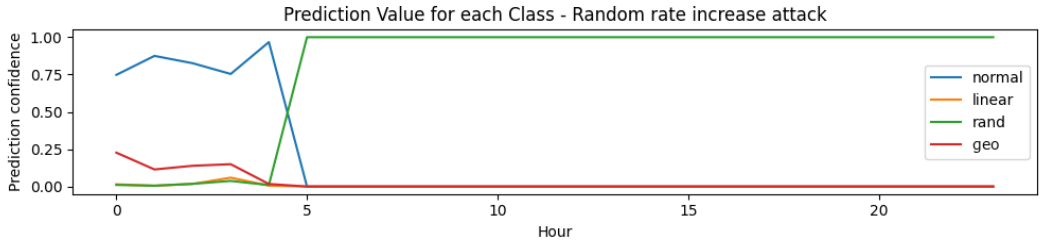

Fig. 6.12 Prediction value of each class for 24 hours in attack scenario

a robust system for further attack detection as it could lead to blacklisting entire regions from access to an application.

Our proposed detection system using DoWNet is successful in providing early warning to an application owner. It is also lightweight in that it is only ever analysing one image. Since that one image is continuously updated, there is no need for storage of multiple heat maps.

6.5.2 Effective Use of DoWNet

DoWNet need only be used where the pixel streaming analysis takes place. We recommend that it be deployed also in a serverless function to be executed every hour. It may gather request traffic data via the serverless platform's inbuilt accessor functions. However, in order to avoid vendor lock-in, a small helper script can be appended to each function in the application that will update a document containing the number of invocations of the functions. This document should then be accessible to the pixel streaming code. We advocate for the deployment of DoWNet on a serverless function operating within the existing application's serverless infrastructure rather than centralising the detection within the firewall offering of the cloud provider. We believe in security driven development, where developers should be coding with security in mind. By providing an open source Python package that is easy to deploy, we are putting the power in the hands of the developer allowing them to be in control of their security without the need to enlist third party services for scrubbing of traffic, storing data etc. This is especially important as serverless computing becomes increasingly popular and is adopted by solo-operator developers as the paradigm of choice when creating their products. A free to use open source package will serve these budget conscious users to secure their functions. DoWNet is a detection system for early warning

of potential attacks. Its value is that it provides objective interpretation of the traffic to the application owner to avoid cases where they may have been fooled into thinking the traffic is normal or are not monitoring it rigorously enough. A detection from DoWNet should be followed up with appropriate mitigation depending on the serverless platform in operation and the requirements of the application owner. Mitigation can vary from shutting the functions down to wide-scale IP address blocking. The former, however, is effectively DoS, and the latter may be difficult if a large botnet with changing IP addresses is being used.

6.6 Summary

In summary, we have applied the novel method of representing request traffic as heat maps for training a DoW classification algorithm. This method overcomes the hurdles obstructing the application of ML to DoW detection that are, enormous volumes of data and difficulty in distinguishing individual requests as malicious or benign. The CNN we trained for the purpose of DoW classification, DoWNet, achieved consistent results in detecting hypothetical attacks proposed [129], with accuracy, F1 score, precision, and recall of approximately 97%. As such, we answered *RQ3* "Can an attack that so closely mimics regular traffic be detected and mitigated against?". This model was then successfully applied to a system that streams incoming request traffic numbers for the re-drawing of heat maps with respect to historical values. These heat maps undergo analysis by DoWNet which will report on the status of the traffic at a given hour to serve as an objective early warning system for application owners. This system was then shown to be capable of being deployed within a serverless function. This lowers the barrier of difficulty in implementing such a detection system on an existing application as it would not require any adaption of existing functions.

CHAPTER 7

Conclusion

This chapter concludes the work presented with a summary of the main contributions and further discussions on the impact, limitations, and potential future work arising from this thesis.

7.1 Summary of Research Contributions

In summary the main contributions of this thesis are as follows:

7.1.1 Formal Definition and Analysis of DoW

As stated throughout Chapter 3, prior to our publication on DoW there was no formal discussion on DoW in the academic literature, discourse was limited to blogs and technology forums. By providing a formal definition we have opened the subject to the wider academic community and have seen interest in the topic rise through mainstream media [56] and research papers directly citing our work as the de facto definition of the attack [58, 57].

Beyond the definition, we also provided the most in depth analysis of theoretical damage potential on the three largest serverless platforms. We also theorise a new vector of attack we term a *leech* attack, which we believe is a true threat to

serverless applications. This work addresses RQ1 and confirms our first hypothesis. We hope this discussion on DoW will encourage developers to be actively aware of the threat and attempt to mitigate against it.

7.1.2 Creation of a System that Addresses the Lack of Data for Expanded Research

This research has been an opportunity for the preemptive analysis of a looming cyber threat. To this end, expanded research on the topic requires the use of synthetic datasets. Our system, DoWTS, presented in Chapter 5, solves that issue. Our heuristic for modelling normal traffic data serves as a good baseline for any future research on advanced attack strategies until such time that historical traffic logs are available. The creation of DoWTS confirms our second hypothesis and answers RQ2.

Also in Chapter 5, we describe how an isolated analogue of a commercial platform can be devised for testing of attack vectors and deployable mitigation solutions. This system is invaluable for further research without the need to inflict real financial damage on oneself.

7.1.3 Development of a Novel Means of Attack Detection with a Proposed Deployment Method

The final major contribution of this thesis is the culmination of all the prior work to the development of a proposed mitigation approach to DoW. This system blends the realms of cyber security with machine learning through a novel approach to attack visualisation. The use of image representations of application traffic allowed for the training of DoWNet, a CNN that can distinguish between expected normal traffic and three early hypothesised attack vectors of DoW. This detection system addresses the concerns raised in our definition of DoW of the new vector of attack, the *leech* attack. We believe this is incredibly important as it is a first solution to this unknown threat. DoWTS affirms RQ3 and confirms our third and final hypothesis.

This model can be deployed to a serverless function that sits within the application structure, requiring little processing or cost overhead to provide vastly improved protection.

7.2 Limitations

Despite the contributions outlined above, this work does have limitations, as outlined below:

7.2.1 Synthetic Data Robustness

The presented system for generating synthetic usage traffic, DoWTS, utilises a heuristic for best approximation of traffic loads over a 24 hour period given a known rough usage base. The heuristic is based on examples of traffic to real ecommerce websites, where there were clear trends in usage, with dips at night and peaks during the day. We concluded that our synthetic data closely follows this example data. However, the load patterns DoWTS replicates are not conducive of an exhaustive sample of *normal* traffic. Therefore, in the training of DoWNet, we are aware that its classification of *normal* traffic may not be sufficient.

We believe that DoWTS provides a perfectly usable dataset for development of mitigation strategies. However, in order to create more robust classifiers, reliance on said synthetic data is not ideal. Instead, a wider sample of example data of normal traffic patterns is required. Upon which the use of more complex forms of synthetic data generation such as the previously discussed GAN and VAE should be considered.

7.2.2 Willingness of Data Sharing

Building on the previous limitation, datasets of normal usage on a serverless application and suspected DoW attacks are required for further research. Normal usage data may become more available over time, just as there are traffic logs of traditional web applications.

However, the biggest hurdle in terms of data acquisition will be recordings of suspected attacks. Unlike DoS where customers are impacted by the attack and therefore a statement must be made by the service owner, DoW only affects the owner. As such there would be no requirement to publicly disclose that they were a victim as this may make investors and other stakeholders uneasy. Data would need to be willingly published for the research community in a selfless pursuit of furthering knowledge on the subject.

7.2.3 Feasibility of Attack

The greatest limitation of this research is the double edged nature of what makes it unique. By being a preemptive analysis of an unrealised attack, it brings into question the viability of the attack entirely. This research has categorically proven that the attack is possible and is indeed a threat to serverless computing if it is executed correctly. However, with a lack of historical examples, the feasibility of the attack comes into play. We are of the opinion that regardless of this limitation, research on the topic should be conducted so that if the attack is realised, there is a body of work to fall back on when devising means of defence. We have outlined potential motivations to conduct DoW and rudimentary attack vectors that bad actors may utilise. Also, it is important to look at existing attack types like DDoS. While no one has publicly admitted to performing a *DDoS* attack on serverless functions to cause a DoW, it is entirely feasible to have happened. Until there is a major occurrence made public, this limitation will remain at the forefront of future researchers minds.

7.3 Future Work

Following from the investigations in this thesis, there are a number of promising avenues for future research:

7.3.1 The Future of Denial of Wallet

The initial investigation into DoW presented in the thesis serves as the starting point of continued research on the attack. We have covered the basics of how it may manifest itself and a first proposal of how to address it. However, as we have alluded to in Chapter 3 when defining DoW, this research focused on serverless function abuse only. Many aspects of cloud computing are moving to a *pay-per-invocation* pricing model. Therefore, there is an ever increasing number of attack surfaces for DoW to hit. We believe the future of DoW research should build on our specific serverless model and begin to encompass protection for these other services.

The means of attack should also be expanded beyond HTTP trigger abuse. Work has already begun on other vectors of attack [58], and until attacks begin

appearing in the wild, researchers must put on their black hats and theorise how pay-per-use systems could be abused in order to preemptively protect them.

Continued research will need to migrate to actual deployments of applications. Our isolated zone for testing serves as a great starting point for mitigation system validation. However, before any product of research is accepted for use in the field, it will need to prove itself on commercial platforms. To this end, collaboration with the commercial platforms is imperative so that the financial damage caused through testing is not a factor that would cause research teams to avoid this area.

Finally, the issue of data acquisition must be addressed. If there are suspected attacks taking place they must be reported. Whether an anonymous means of reporting is required or collaboration with large companies utilising serverless computing needs to be setup, somehow the veil must be lifted on DoW in order for the research community to properly tackle the attack.

7.3.2 Image Processing and Classification in Cybersecurity

This research, along with the related work presented in Chapter 2, has shown that image classifiers are a powerful tool for cyber threat detection. A great deal of information can be conveyed within an image and given the current spike in the effectiveness of image classification algorithms (as demonstrated by the ILSVRC), this method of threat detection shows strong promise.

Image classification proved especially powerful in our use case as we were translating log files that would total over 20GB in size to images that are barely over a kilobyte. This drastically more efficient means of storing information would prove useful for analysis of other attacks, especially those that execute over long time spans. It may also prove effective in related fields such as bot detection, click fraud and espionage detection with potential additions to the image generation process such as Gramian angular field generation [140].

As image recognition packages become more streamlined, it brings with it the possibility of utilising image classification cyber threat detectors in low processing power deployments, such as serverless functions. This may be an avenue that could disrupt the current cyber security market where clients are reliant on outsourcing their data to an external service for processing, rather than running their security solution on their hardware or cloud.

7.3.3 Beyond Computer Science

Our research is primarily focused on the technical side of DoW, both in its execution and damage it may cause. We alluded to potential motivations for conducting such an attack and theoretical ramifications for the damage it can cause. However, this topic may go beyond the remit of computer science.

From a psychology point of view, further research may be conducted on:

- The mindset of an application owner in distinguishing between an attack or natural growth of traffic. At what point does doubt set in? Are there governing factors in how a victim can be fooled into believing a spike in traffic is legitimate?
- Continued discussion on the motivation behind such an attack.
- How can an attacker use the psychology of the victim to better design attack vectors that maximise damage?

From a business and economics perspective, further research on the ramifications of DoW attacks of varying success on a business would add to discussion on this topic. We have demonstrated the immediate damage of DoW, though there is certainly more at stake when finances are involved.

7.4 Final Remarks

As serverless computing and other pay-per-invocation services become the norm in application development, the DoW attack may become more and more inevitable. The community has been slow to accept this either because it was assumed traditional mitigation methods for similar attacks such as DoS were enough, or they simply believe it is an operational cost. However, this thesis has demonstrated that there is more to DoW than initially thought, and by bringing it into the academic community we hope it will serve as a starting point for any future research on the topic.

References

[1] Eric McCullough, Razib Iqbal, and Ajay Katangur. Analysis of machine learning techniques for lightweight ddos attack detection on iot networks. In *International Conference on Forthcoming Networks and Sustainability in the IoT Era*, pages 96–110. Springer, 2020.

[2] James Beswick. Load testing a web application's serverless backend, 2020. URL https://aws.amazon.com/blogs/compute/load-testing-a-web-applications-serverless-backend/.

[3] Anina Ot. The serverless computing market in 2022, May 2022. URL https://www.enterprisestorageforum.com/cloud/serverless-computing-market/.

[4] Fernando J Corbató, Marjorie Merwin-Daggett, and Robert C Daley. An experimental time-sharing system. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, pages 335–344, 1962.

[5] Martin Greenberger, Herbert A. Simon, C. P. Snow, Robert C. Sprague, Norbert Wiener, and John McCarthy. *Time-Sharing Computer Systems*, page 236–236. M.I.T. Press, Massachusetts Insitute of Technology, 1962.

[6] Jayachander Surbiryala and Chunming Rong. Cloud computing: History and overview. In *2019 IEEE Cloud Summit*, pages 1–7. IEEE, 2019.

[7] RedHat. Types of cloud computing, Jul 2022. URL https://www.redhat.com/en/topics/cloud-computing/public-cloud-vs-private-cloud-and-hybrid-cloud.

[8] Ganesh Venkitachalam and Tzi-cker Chiueh. High performance common gateway interface invocation. In *Proceedings 1999 IEEE Workshop on Internet Applications (Cat. No. PR00197)*, pages 4–11. IEEE, 1999.

[9] Ken Fromm. Why the future of software and apps is serverless, Oct 2012. URL https://readwrite.com/why-the-future-of-software-and-apps-is-serverless/.

[10] CloudFlare. Why use serverless computing? | pros and cons of serverless. URL https://www.cloudflare.com/learning/serverless/why-use-serverless/.

[11] Mike Gualtieri. I don't want devops. i want noops., Feb 2011. URL https://www.forrester.com/blogs/11-02-07-i_dont_want_devops_i_want_noops/.

[12] AWS. Aws lambda, . URL https://aws.amazon.com/lambda/.

[13] Stefanie James, Chris Harbron, Janice Branson, and Mimmi Sundler. Synthetic data use: exploring use cases to optimise data utility. *Discover Artificial Intelligence*, 1(1):15, 2021.

[14] Gilad David Maayan. Will synthetic data introduce ethical challenges for ml engineers?, Jul 2022. URL https://towardsdatascience.com/will-synthetic-data-introduce-ethical-challenges-for-ml-engineers-b2608139d27f.

[15] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A Bharath. Generative adversarial networks: An overview. *IEEE signal processing magazine*, 35(1):53–65, 2018.

[16] Maayan Frid-Adar, Eyal Klang, Michal Amitai, Jacob Goldberger, and Hayit Greenspan. Synthetic data augmentation using gan for improved liver lesion classification. In *2018 IEEE 15th international symposium on biomedical imaging (ISBI 2018)*, pages 289–293. IEEE, 2018.

[17] Christopher Bowles, Liang Chen, Ricardo Guerrero, Paul Bentley, Roger Gunn, Alexander Hammers, David Alexander Dickie, Maria Valdés Hernández, Joanna Wardlaw, and Daniel Rueckert. Gan augmentation: Augmenting training data using generative adversarial networks. *arXiv preprint arXiv:1810.10863*, 2018.

[18] Vinay Kukreja, Deepak Kumar, Amandeep Kaur, et al. Gan-based synthetic data augmentation for increased cnn performance in vehicle number plate recognition. In *2020 4th international conference on electronics, communication and aerospace technology (ICECA)*, pages 1190–1195. IEEE, 2020.

[19] Jun-Hyung Kim and Youngbae Hwang. Gan-based synthetic data augmentation for infrared small target detection. *IEEE Transactions on Geoscience and Remote Sensing*, 60:1–12, 2022.

[20] Lei Xu, Maria Skoularidou, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. Modeling tabular data using conditional gan. *Advances in Neural Information Processing Systems*, 32, 2019.

[21] Diederik P. Kingma and Max Welling. An introduction to variational autoencoders. *Foundations and Trends® in Machine Learning*, 12(4):307–392, 2019. doi: 10.1561/2200000056. URL https://doi.org/10.1561%2F2200000056.

[22] Xianxu Hou, Linlin Shen, Ke Sun, and Guoping Qiu. Deep feature consistent variational autoencoder. In *2017 IEEE winter conference on applications of computer vision (WACV)*, pages 1133–1141. IEEE, 2017.

[23] Gianluigi Greco, Antonella Guzzo, and Giuseppe Nardiello. Fd-vae: A feature driven vae architecture for flexible synthetic data generation. In *Database and Expert Systems Applications: 31st International Conference, DEXA 2020, Bratislava, Slovakia, September 14–17, 2020, Proceedings, Part I*, pages 188–197. Springer, 2020.

[24] Zhiqiang Wan, Yazhou Zhang, and Haibo He. Variational autoencoder based synthetic data generation for imbalanced learning. In *2017 IEEE symposium series on computational intelligence (SSCI)*, pages 1–7. IEEE, 2017.

[25] Drake Cullen, James Halladay, Nathan Briner, Ram Basnet, Jeremy Bergen, and Tenzin Doleck. Evaluation of synthetic data generation techniques in the domain of anonymous traffic classification. *IEEE Access*, 10:129612–129625, 2022.

[26] Qi Liu, Miltiadis Allamanis, Marc Brockschmidt, and Alexander Gaunt. Constrained graph variational autoencoders for molecule design. *Advances in neural information processing systems*, 31, 2018.

[27] Siddhartha Chib. Markov chain monte carlo methods: computation and inference. *Handbook of econometrics*, 5:3569–3649, 2001.

[28] Jeremy Charlier, Aman Singh, Gaston Ormazabal, Radu State, and Henning Schulzrinne. Syngan: Towards generating synthetic network attacks using gans. *arXiv preprint arXiv:1908.09899*, 2019.

[29] TD Ovasapyan, VD Danilov, and Dmitry A Moskvin. Application of synthetic data generation methods to the detection of network attacks on internet of things devices. *Automatic Control and Computer Sciences*, 55(8):991–998, 2021.

[30] Carlos Garcia Cordero, Emmanouil Vasilomanolakis, Aidmar Wainakh, Max Mühlhäuser, and Simin Nadjm-Tehrani. On generating network traffic datasets with synthetic attacks for intrusion detection. *ACM Transactions on Privacy and Security (TOPS)*, 24(2):1–39, 2021.

[31] Shengzhe Xu, Manish Marwah, Martin Arlitt, and Naren Ramakrishnan. Stan: Synthetic network traffic generation with generative neural models. In *International Workshop on Deployable Machine Learning for Security Defense*, pages 3–29. Springer, 2021.

[32] Jasek Roman, Benda Radek, Vala Radek, and Sarga Libor. Launching distributed denial of service attacks by network protocol exploitation. In *Proceedings of the 2nd international conference on Applied informatics and computing theory*, pages 210–216, 2011.

[33] Christos Douligeris and Aikaterini Mitrokotsa. Ddos attacks and defense mechanisms: classification and state-of-the-art. *Computer networks*, 44(5): 643–666, 2004.

[34] Christofer Hoff. Cloud computing security: From ddos (distributed denial of service) to edos (economic denial of sustainability), 2008. URL https://www.rationalsurvivability.com/blog/2008/11/cloud-computing-security-from-ddos-distributed-denial-of-service-to-edos-economic-denial-of-sustainability/.

[35] Kenneth C Wilbur and Yi Zhu. Click fraud. *Marketing Science*, 28(2):293–308, 2009.

[36] Firebrand. Bot traffic detection method teases real website traffic from fake, 2017. URL https://firebrand.net/blog/bot-traffic-detection-tool/,urldate=2020-07-01.

[37] Cornel Barna, Mark Shtern, Michael Smit, Vassilios Tzerpos, and Marin Litoiu. Model-based adaptive dos attack mitigation. In *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 119–128. IEEE, 2012.

[38] Madarapu Naresh Kumar, P Sujatha, Vamshi Kalva, Rohit Nagori, Anil Kumar Katukojwala, and Mukesh Kumar. Mitigating economic denial of sustainability (edos) in cloud computing using in-cloud scrubber service. In *2012 Fourth international conference on computational intelligence and communication networks*, pages 535–539. IEEE, 2012.

[39] Soon Hin Khor and Akihiro Nakao. spow: On-demand cloud-based eddos mitigation mechanism. In *HotDep (Fifth Workshop on Hot Topics in System Dependability)*, 2009.

[40] Joseph Idziorek and Mark Tannian. Exploiting cloud utility models for profit and ruin. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 33–40. IEEE, 2011.

[41] Mohammed H Sqalli, Fahd Al-Haidari, and Khaled Salah. Edos-shield-a two-steps mitigation technique against edos attacks in cloud computing. In *2011 Fourth IEEE international conference on utility and cloud computing*, pages 49–56. IEEE, 2011.

[42] Nir Kshetri. The economics of click fraud. *IEEE Security & Privacy*, 8(3): 45–53, 2010.

[43] Ana. Bot baseline fraud in digital advertising 2016-2017. Report, 2017. URL https://www.ana.net/getfile/25093.

[44] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. Valve: Securing function workflows on serverless computing platforms. In *Proceedings of The Web Conference 2020*, pages 939–950, 2020.

[45] OWASP. Owasp api security project. Report, 2019. URL https://owasp.org/www-project-api-security/.

[46] Jun-Ming Su, Ming-Hua Cheng, Xin-Jie Wang, and Shian-Shyong Tseng. A scheme to create simulated test items for facilitating the assessment in web security subject. In *2019 Twelfth International Conference on Ubi-Media Computing (Ubi-Media)*, pages 306–309. IEEE, 2019.

[47] Nobuaki Maki, Ryotaro Nakata, Shinichi Toyoda, Yosuke Kasai, Sanggyu Shin, and Yoichi Seto. An effective cybersecurity exercises platform cyexec and its training contents. *International Journal of Information and Education Technology*, 10(3):215–221, 2020.

[48] Muhammad Idris, Iwan Syarif, and Idris Winarno. Development of vulnerable web application based on owasp api security risks. In *2021 International Electronics Symposium (IES)*, pages 190–194. IEEE, 2021.

[49] Doug Dooley. Serverless, shadow APIs and Denial of Wallet attacks, 2019. URL https://www.helpnetsecurity.com/2019/03/29/serverless-challenges/.

[50] Roman Sachenko. Severe Truth About Serverless Security and Ways to Mitigate Major Risks, 2020. URL https://hackernoon.com/severe-truth-about-serverless-security-and-ways-to-mitigate-major-risks-cd3i3x6f.

[51] Adnan Rahic. Fantastic Serverless security risks, and where to find them, 2018. URL https://www.serverless.com/blog/fantastic-serverless-security-risks-and-where-to-find-them/.

[52] Pursec. The ten most critical security risks in serverless architecture, 2018. URL https://www.puresec.io/hubfs/SAS-Top10-2018/PureSec-SASTop10-2018.pdf.

[53] OWASP. OWASP Top 10 (2017) Interpretation for Serverless. Technical report. URL https://owasp.org/www-pdf-archive/OWASP-Top-10-Serverless-Interpretation-en.pdf.

[54] AWS. Security overview of aws lambda. Report, 2021.

[55] AWS. Serverless applications lens aws well-architected framework. Report, 2019.

[56] Thomas Claburn. Not saying you should but we're told it's possible to land serverless app a '$40k/month bill using a 1,000-node botnet', Apr 2021. URL https://www.theregister.com/2021/04/21/denial_of_wallet/.

[57] Dimitar Mileski and Hristina Mihajloska. Distributed Denial of Wallet Attack on Serverless Pay-as-you-go Model. In *2022 30th Telecommunications Forum (TELFOR)*, pages 1–4. IEEE, 2022.

[58] Junxian Shen, Han Zhang, Yantao Geng, Jiawei Li, Jilong Wang, and Mingwei Xu. Gringotts: Fast and Accurate Internal Denial-of-Wallet Detection for Serverless Computing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, pages 2627–2641, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 978-1-4503-9450-5. doi: 10.1145/3548606.3560629. URL https://doi.org/10.1145/3548606.3560629. event-place: Los Angeles, CA, USA.

[59] Kamran Shaukat, Suhuai Luo, Vijay Varadharajan, Ibrahim A Hameed, and Min Xu. A survey on machine learning techniques for cyber security in the last decade. *IEEE Access*, 8:222310–222354, 2020.

[60] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. In *2018 IEEE symposium on security and privacy (SP)*, pages 19–35. IEEE, 2018.

[61] ImageNet. Imagenet large scale visual recognition challenge (ilsvrc). URL https://image-net.org/challenges/LSVRC/index.php.

[62] Image classification. URL https://paperswithcode.com/task/image-classification. Accessed on 24.11.2022.

[63] Jefkine Kafunah. Backpropagation in convolutional neural networks, Sep 2016. URL https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/.

[64] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[65] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2015.

[66] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[67] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[68] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2016.

[69] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[70] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[71] Quamar Niyaz, Weiqing Sun, and Ahmad Y Javaid. A deep learning based ddos detection system in software-defined networking (sdn). *arXiv preprint arXiv:1611.07400*, 2016.

[72] Zecheng He, Tianwei Zhang, and Ruby B Lee. Machine learning based ddos attack detection from source side in cloud. In *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*, pages 114–120. IEEE, 2017.

[73] S Shanmuga Priya, M Sivaram, D Yuvaraj, and A Jayanthiladevi. Machine learning based ddos detection. In *2020 International Conference on Emerging Smart Computing and Informatics (ESCI)*, pages 234–237. IEEE, 2020.

[74] Ili Ko, Desmond Chambers, and Enda Barrett. Feature dynamic deep learning approach for ddos mitigation within the isp domain. *International Journal of Information Security*, 19(1):53–70, 2020.

[75] Markus-Go. Bonesi. URL https://github.com/Markus-Go/bonesi.

[76] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath. Malware images: Visualization and automatic classification. In *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, VizSec '11, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306799. doi: 10.1145/2016904.2016908. URL https://doi.org/10.1145/2016904.2016908.

[77] Jhu-Sin Luo and Dan Chia-Tien Lo. Binary malware image classification using machine learning with local binary pattern. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 4664–4667. IEEE, 2017.

[78] Scott Freitas, Rahul Duggal, and Duen Horng Chau. Malnet: A large-scale cybersecurity image database of malicious software. *arXiv preprint arXiv:2102.01072*, 2021.

[79] Ahmad Azab and Mahmoud Khasawneh. Msic: malware spectrogram image classification. *IEEE Access*, 8:102007–102021, 2020.

[80] Wei Wang, Ming Zhu, Xuewen Zeng, Xiaozhou Ye, and Yiqiang Sheng. Malware traffic classification using convolutional neural network for representation learning. In *2017 International conference on information networking (ICOIN)*, pages 712–717. IEEE, 2017.

[81] Shayan Taheri, Milad Salem, and Jiann-Shiun Yuan. Leveraging image representation of network traffic data and transfer learning in botnet detection. *Big Data and Cognitive Computing*, 2(4), 2018. ISSN 2504-2289. doi: 10.3390/bdcc2040037. URL https://www.mdpi.com/2504-2289/2/4/37.

[82] Sebastian Garcia, Martin Grill, Jan Stiborek, and Alejandro Zunino. An empirical comparison of botnet detection methods. *computers & security*, 45:100–123, 2014.

[83] Daniel Kelly, Frank G Glavin, and Enda Barrett. Denial of wallet—defining a looming threat to serverless computing. *Journal of Information Security and Applications*, 60:102843, 2021.

[84] Guillame Ross. richorama denial of wallet attack!, 07/05/2013 2013.

[85] Rachel Kaser. Protesters flood dallas police app with k-pop videos, Jun 2020. URL https://thenextweb.com/news/protesters-flood-dallas-police-app-kpop-videos.

[86] Scott Piper. Denial of wallet attacks on aws, Jun 2020. URL https://summitroute.com/blog/2020/06/08/denial_of_wallet_attacks_on_aws/.

[87] AWS. Aws waf, . URL https://aws.amazon.com/waf/.

[88] Google. Cloud armor network security | google cloud armor. URL https://cloud.google.com/armor.

[89] Microsoft. Azure web application firewall (waf): Microsoft azure. URL https://azure.microsoft.com/en-us/products/web-application-firewall/.

[90] AWS. Amazon API Gateway quotas and important notes, . URL https://docs.aws.amazon.com/apigateway/latest/developerguide/limits.html.

[91] Saman Taghavi Zargar, James Joshi, and David Tipper. A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks. *IEEE communications surveys & tutorials*, 15(4):2046–2069, 2013.

[92] Nikhil Tripathi and Neminath Hubballi. Slow rate denial of service attacks against http/2 and detection. *Computers & security*, 72:255–272, 2018.

[93] Molly Sauter. "loic will tear us apart" the impact of tool design and media portrayals in the success of activist ddos attacks. *American Behavioral Scientist*, 57(7):983–1007, 2013.

[94] Onur Varol, Emilio Ferrara, Clayton Davis, Filippo Menczer, and Alessandro Flammini. Online human-bot interactions: Detection, estimation, and characterization. In *Proceedings of the international AAAI conference on web and social media*, volume 11, pages 280–289, 2017.

[95] Salman Aslam. Twitter by the Numbers: Stats, Demographics & Fun Facts, 2020. URL https://www.omnicoreagency.com/twitter-statistics/.

[96] Qiang Cao, Michael Sirivianos, Xiaowei Yang, and Tiago Pregueiro. Aiding the detection of fake accounts in large scale social online services. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 197–210, 2012.

[97] Chao Yang, Robert Harkreader, Jialong Zhang, Seungwon Shin, and Guofei Gu. Analyzing spammers' social networks for fun and profit: a case study of cyber criminal ecosystem on twitter. In *Proceedings of the 21st international conference on World Wide Web*, pages 71–80, 2012.

[98] Jinyuan Jia, Binghui Wang, and Neil Zhenqiang Gong. Random walk based fake account detection in online social networks. In *2017 47th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 273–284. IEEE, 2017.

[99] Abdulrahman Alarifi, Mansour Alsaleh, and AbdulMalik Al-Salman. Twitter turing test: Identifying social machines. *Information Sciences*, 372:332–346, 2016.

[100] Buket Erşahin, Özlem Aktaş, Deniz Kılınç, and Ceyhun Akyol. Twitter fake account detection. In *2017 International Conference on Computer Science and Engineering (UBMK)*, pages 388–392. IEEE, 2017.

[101] Mayra Rorsario Fuentes. Shifts in underground markets past, present and future. Report, Trend Micro, 2020. URL https://documents.trendmicro.com/assets/white_papers/wp-shifts-in-the-underground.pdf.

[102] AWS. Aws case study: The seattle times. Report, . URL https://aws.amazon.com/solutions/case-studies/the-seattle-times/.

[103] Rohit Akiwatkar. 10 AWS Lambda Use Cases to Start Your Serverless Journey. URL https://www.simform.com/serverless-examples-aws-lambda-use-cases/.

[104] Nidhin Kumar. Resize an Image in AWS S3 Using a Lambda Function, 2019. URL https://levelup.gitconnected.com/resize-an-image-in-aws-s3-using-lambda-function-dc386afd4128.

[105] John Lim. Here's how much the iPhone's camera has changed over the years, 2019. URL https://sea.mashable.com/tech/6504/heres-how-much-the-iphones-camera-has-changed-over-the-years.

[106] Hasan Yasar. Experiment: Sizing exposed credentials in github public repositories for ci/cd. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 143–143. IEEE, 2018.

[107] AWS. Wild rydes, . URL https://aws.amazon.com/getting-started/hands-on/build-serverless-web-app-lambda-apigateway-s3-dynamodb-cognito/.

[108] James Beswick. Building a location-based, scalable, serverless web app, 2020. URL https://aws.amazon.com/blogs/compute/building-a-location-based-scalable-serverless-web-app-part-1/.

[109] Daniel Kelly, Frank Glavin, and Enda Barrett. Serverless computing: Behind the scenes of major platforms. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 304–312. IEEE, 2020.

[110] Cui Yan. How long does AWS Lambda keep your idle functions around before a cold start?, 2017. URL https://read.acloud.guru/how-long-does-aws-lambda-keep-your-idle-functions-around-before-a-cold-start-bf715d3b810.

[111] Cui Yan. How does language, memory and package size affect cold starts of AWS Lambda?, 2017. URL https://read.acloud.guru/does-coding-language-memory-or-package-size-affect-cold-starts-of-aws-lambda-a15e26d12c76.

[112] Shilkov Mikhail. Comparison of Cold Starts in Serverless Functions across AWS, Azure, and GCP, 2019. URL https://mikhail.io/serverless/coldstarts/big3/.

[113] Byrro Renato. Can We Solve Serverless Cold Starts?, 2019. URL https://dashbird.io/blog/can-we-solve-serverless-cold-starts/.

[114] Voijta Robert. AWS journey — API Gateway & Lambda & VPC performance, 2016. URL https://www.zrzka.dev/2016/10/30/aws-journey-api-gateway-lambda-vpc-performance.html.

[115] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with {OpenLambda}. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[116] Garrett McGrath and Paul R Brenner. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410. IEEE, 2017.

[117] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. Serverless computing: An investigation of factors influencing microservice performance. In *2018 IEEE international conference on cloud engineering (IC2E)*, pages 159–169. IEEE, 2018.

[118] Carlos M Aderaldo, Nabor C Mendonça, Claus Pahl, and Pooyan Jamshidi. Benchmark requirements for microservices architecture research. In *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, pages 8–13. IEEE, 2017.

[119] Sunil Kumar Mohanty, Gopika Premsankar, Mario Di Francesco, et al. An evaluation of open source serverless computing frameworks. *CloudCom*, 2018:115–120, 2018.

[120] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[121] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, 2018.

[122] Daniel Kelly. Serverless computing behind the scenes of major platforms data set, . URL https://github.com/psykodan/Serverless-Computing-Data-Set.

[123] Intel® 64 and IA-32 ArchitecturesSoftware Developer's Manual, 2019. URL https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

[124] Shilkov Mikhail. Serverless: Cold Start War, 2018. URL https://mikhail.io/2018/08/serverless-cold-start-war/.

[125] Ping-Min Lin and Alex Glikson. Mitigating cold starts in serverless platforms: A pool-based approach. *arXiv preprint arXiv:1903.12221*, 2019.

[126] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research advances in cloud computing*, pages 1–20. Springer, 2017.

[127] Google Cloud. Reading and writing temporary files. URL https://cloud.google.com/appengine/docs/standard/python3/using-temp-files.

[128] GNUPlot. Gnuplot acsplines. URL https://gnuplot.sourceforge.net/docs_4.2/node125.html.

[129] Daniel Kelly, Frank G Glavin, and Enda Barrett. Dowts–denial-of-wallet test simulator: Synthetic data generation for preemptive defence. *Journal of Intelligent Information Systems*, pages 1–24, 2022.

[130] Daniel Kelly. Dowts - denial of wallet test simulator, . URL https://github.com/psykodan/DoWTS.

[131] Michael Kechinov. ecommerce events history in cosmetics shop, 2020. URL https://www.kaggle.com/datasets/mkechinov/ecommerce-events-history-in-cosmetics-shop.

[132] Michael Kechinov. ecommerce events history in electronics store, 2021. URL https://www.kaggle.com/datasets/mkechinov/ecommerce-events-history-in-electronics-store.

[133] REES46 Technologies. Open cdp. URL https://rees46.com/en/open-cdp.

[134] Data Science Campus. Synth gauge. URL https://github.com/datasciencecampus/synthgauge.

[135] Scipy. Kolmogorov smirnov test, 2022. URL https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ks_2samp.html.

[136] Scipy. Wasserstein distance, 2022. URL https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.wasserstein_distance.html.

[137] Scipy. Jensen shannon distance, 2022. URL https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.jensenshannon.html.

[138] Junxian Shen, Han Zhang, Yantao Geng, Jiawei Li, Jilong Wang, and Mingwei Xu. Gringotts: Fast and accurate internal denial-of-wallet detection for serverless computing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 2627–2641, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394505. doi: 10.1145/3548606.3560629. URL https://doi.org/10.1145/3548606.3560629.

[139] Keras documentation: Keras applications. URL https://keras.io/api/applications/. Accessed on 24.11.2022.

[140] PYTS. Single gramian angular field. URL https://pyts.readthedocs.io/en/stable/auto_examples/image/plot_single_gaf.html.

APPENDIX A

DoWTS Normal Traffic Synthesis

A.1 Methodology

Baseline traffic is generated by a random number generator using a Poisson distribution, where the *users per time step* value is used as lambda in equation A.1. The resulting traffic is noisy and random about that value (Figure A.2).

$$P(x) = \frac{e^{-\lambda}\lambda^{x}}{x!} \tag{A.1}$$

Through visual analysis of the e-commerce datasets' number of requests per hour over the course of a month long period, we observed a number of patterns in the data (Figure A.1). Firstly, there appeared to be fluctuations in traffic amounts roughly every seven days. We can infer that this corresponds to increased traffic on weekends. To achieve this fluctuation, a sinusoidal pattern with a period of seven days was added to the Poisson distribution traffic (Figure A.3). This subtly causes the traffic to oscillate.

The next pattern we observed was obvious dips in traffic during late night hours. Again, a sinusoidal pattern was added to the traffic timings in order to achieve stark fluctuations in day/night traffic (Figure A.4).

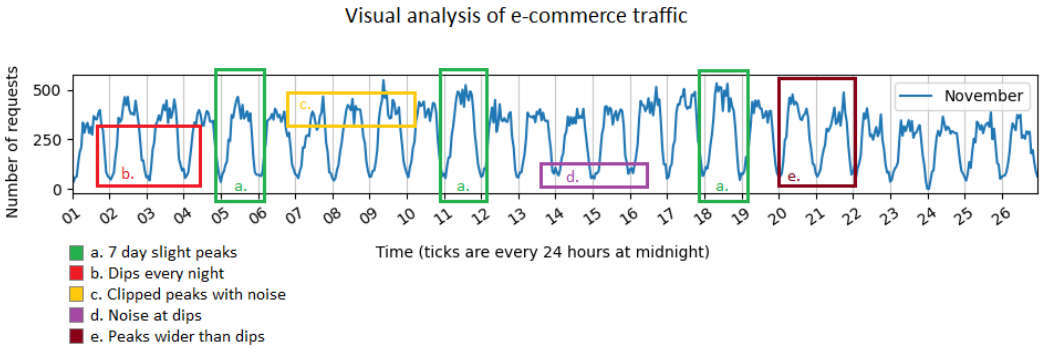

Fig. A.1 Visual analysis of sample data highlighting characteristics replicated by DoWTS.

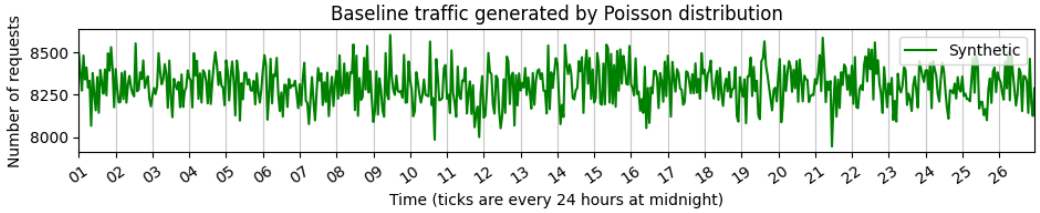

Fig. A.2 Baseline traffic generated by Poisson distribution

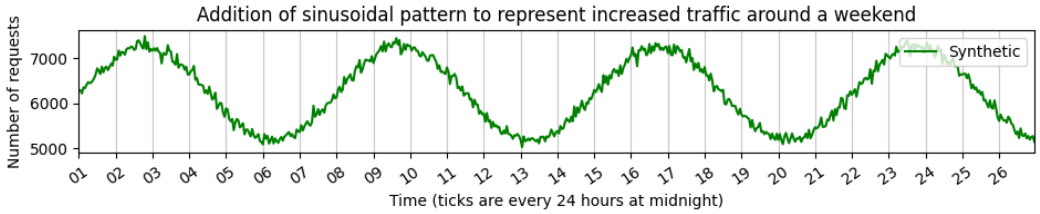

Fig. A.3 Addition of sinusoidal pattern to represent increased traffic around a weekend

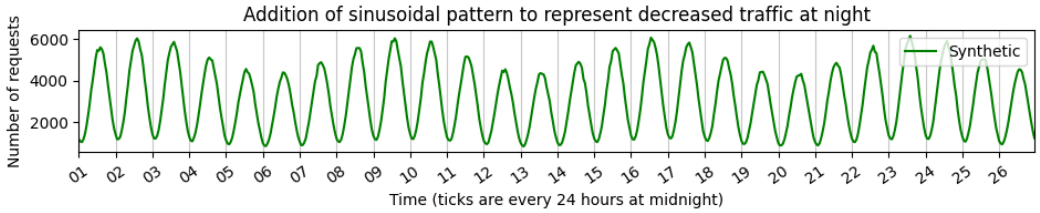

Fig. A.4 The addition of a sinusoidal pattern to represent decreased traffic at night

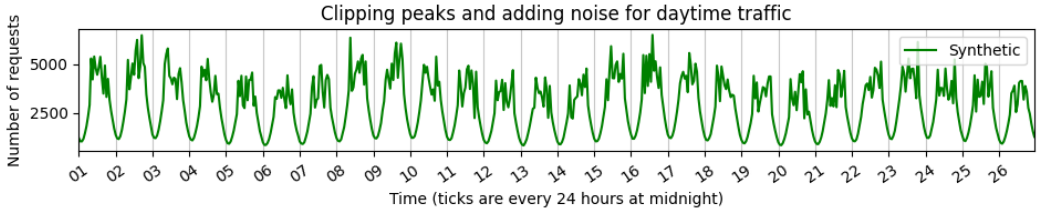

Fig. A.5 Clipping peaks and adding noise for daytime traffic

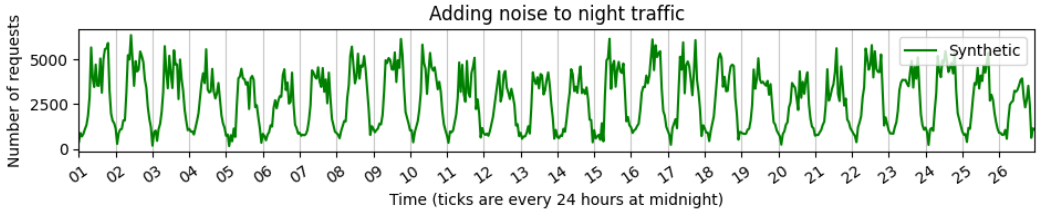

Fig. A.6 Adding noise to night traffic

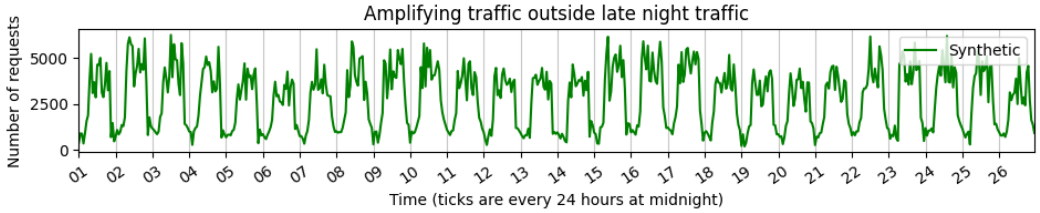

Fig. A.7 Amplifying traffic outside late-night traffic

Daytime traffic did not follow the regular repeating values of a sine wave in the e-commerce data. We observed clipping of the peaks and the traffic at that clipping threshold was noisy. We synthesise this behaviour by setting the traffic to another randomly generated number based on a Poisson distribution when the ratio of *pre-augmented* traffic to *augmented* traffic (that is, traffic before and after the day/night sine augmentation) is less than 2 (Figure A.5).

Similarly, to reduce traffic that followed a perfect sine wave, minor noise was introduced to the troughs of the wave (Figure A.6).

The final observation of our graphical analysis of the ecommerce datasets was that there was less downtime than up time in the traffic, i.e. the peaks are wider than the troughs. By amplifying the traffic between the zones of clipping the peaks and adding noise to the troughs, the effect can be achieved (Figure A.7).

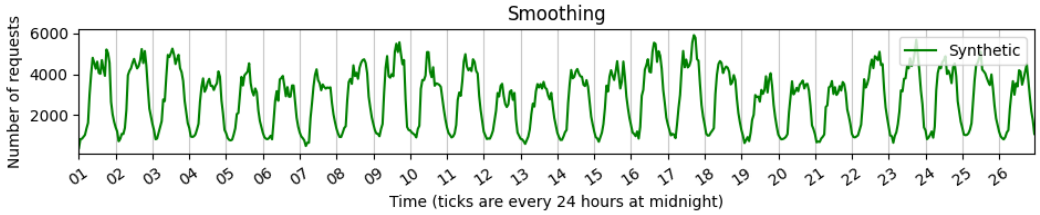

Fig. A.8 Smoothing traffic

Finally, a smoothing algorithm was applied that interpolates the distance between subsequent values should the difference between them be too much. This is required to avoid overly jagged traffic (Figure A.8).