

Neural Architecture Search for Website Fingerprinting

by

Shreya Arun Naik

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2024

© Shreya Arun Naik 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

To protect their online privacy, users often employ encrypted tunnels such as those established by low-latency anonymity networks like Tor. These tunnels enable multi-hop encrypted pathways, concealing the contents and destinations of users’ communications from potential eavesdroppers. However, Tor’s low-latency design preserves the timing and volume properties of network packet exchanges, leading to the creation of distinct traffic patterns, or fingerprints, for each website. These fingerprints render Tor susceptible to traffic analysis attacks, particularly website fingerprinting (WF) attacks, where eavesdroppers exploit machine learning techniques to match traffic patterns to specific websites.

Recent research on WF has shifted from the use of traditional machine learning techniques to the use of deep neural networks (DNNs) in order to devise more precise attacks. However, the development of DNN architectures geared for WF attacks often relies on a trial-and-error adaptation of DNN architectures shown to work well in other domains. Despite its relative success, this heuristic-based development of DNN architectures can overlook unconventional or innovative architectural choices, missing potential breakthroughs.

This thesis seeks to uncover new and effective DNN-based WF attacks by taking advantage of neural architecture search (NAS), a subfield of AutoML which introduces a set of optimization techniques aimed at the automatic generation of effective DNN architectures for specific tasks. For achieving this objective, we design FAWNS, a NAS framework designed for website fingerprinting (and, potentially, for other traffic analysis tasks), which we tightly integrate with Microsoft’s neural network intelligence (NNI) AutoML toolkit. FAWNS automates DNN architectural design by combining various convolutional layers from a predefined search space, enabling systematic exploration and optimization.

Using FAWNS, we conducted comprehensive explorations of NAS search spaces with various search strategies to evaluate the framework’s effectiveness. Our results show that FAWNS-generated DNNs achieve an accuracy comparable to that of state-of-the-art attacks on undefended Tor traffic and significantly higher accuracy on defended traffic, with a 26% accuracy increase on traces protected by the FRONT defense and a 12% increase on WTF-PAD defended traces. Additionally, we find that these models generalize well to unseen data, suggesting that NAS has a compelling potential to improve the effectiveness of WF.

In summary, this thesis describes the design of the FAWNS framework, details how one can leverage it to explore DNNs tailored to WF tasks, and evaluates the effectiveness of the resulting DNN architectures when launching WF attacks, showcasing their robustness against defenses and concept drift.

Acknowledgments

I thank my parents for their unwavering support and my brother for his love and encouragement. I am deeply grateful to my advisor, Prof. Diogo Barradas, for his astute guidance and kindness throughout my master's education. I appreciate my thesis readers, Prof. Ian Goldberg and Prof. Mina Tahmasbi Arashloo, for their valuable comments and feedback that greatly improved this thesis. To my friends who believe in me and consider me worthy of their love, I feel blessed to have you in my life. And to the star I see in all of my night skies, I adore you. Thank you for guiding me with your light.

My gratitude extends to the Cybersecurity and Privacy Institute (CPI) for the Chippie compute facility, the Cryptography, Security, and Privacy (CrySP) lab for the RIPPLE compute facility, and the University of Waterloo and Microsoft for the Azure credits that facilitated this project.

Dedication

*In fond memory of
D. N. Kumbhoji (1936-2023),
my beloved grandfather and my first teacher.*

Table of Contents

Author's Declaration	ii
Abstract	iii
Acknowledgments	iv
Dedication	v
List of Figures	ix
List of Tables	x
1 Introduction	1
1.1 Overview	1
1.2 Contributions	3
1.3 Thesis Outline	4
2 Background and Related Work	5
2.1 Tor	5
2.2 Website Fingerprinting	7
2.2.1 Attacks	7
2.2.2 Defenses	13

2.2.3	Datasets	18
2.3	Neural Architecture Search	19
2.3.1	Search Spaces	21
2.3.2	Search Strategies	23
2.3.3	Performance Estimation Strategies	28
2.4	NAS for WF	29
3	The FAWNS Framework	31
3.1	Extractor	31
3.2	Traffic Analyzer and WF Attacker	33
3.3	NNI's NAS Module	35
3.3.1	Model Space Hub	36
3.3.2	Search Strategy Hub	39
3.3.3	Model Evaluator	40
3.3.4	NAS Experiment Engine	42
3.4	Experiment Manager	42
3.5	Logger	42
4	Implementation of FAWNS Modules	45
4.1	Extractor	45
4.2	Traffic Analyzer and WF Attacker	47
4.3	NAS Module	49
4.3.1	Search Space	49
4.3.2	Search Strategies	52
4.3.3	Evaluation Strategy	53
4.3.4	NAS Experiment Engine	54
4.4	Logger	55

5	Evaluation	56
5.1	Evaluation Methodology	56
5.2	Laboratory Testbed	59
5.3	Optimizing Hyperparameters for NAS Searches	60
5.4	Benchmarking FAWNS on Undefined Traces	62
5.4.1	Accuracy of FAWNS-generated Models	62
5.4.2	Assessing FAWNS-generated Models' Training Costs	63
5.4.3	Integrating Early-stopping on FAWNS-generated Models	64
5.5	Attacking Defended Traces with FAWNS	65
5.5.1	Attacking Defended Traces without Early-stopping	65
5.5.2	Attacking Defended Traces using Early-stopping	67
5.6	Generalizability of FAWNS' DNN Architectures	70
5.6.1	Tolerance to Training on a Downsampled Dataset	70
5.6.2	Tolerance to Concept Drift	72
6	Conclusions	74
	References	77

List of Figures

2.1	A typical Tor circuit	6
2.2	Overview of a website fingerprinting attack setup	8
2.3	The three dimensions of NAS	21
2.4	Examples of chain-structured and cell-based search spaces	22
2.5	An example three-level hierarchical search space	24
3.1	Overview of the FAWNS architecture	32
3.2	An overview of DARTS	37
3.3	Stacking cells in NAS-generated networks	39
3.4	Visualizing optimized DARTS cells	41
3.5	Example of NAS experiment trials on NNI’s web portal	43
3.6	NNI NAS trials visualized on TensorBoard	44
5.1	An overview of FAWNS’ testing infrastructure	58
5.2	Convergence of DARTS_oneshot and Tik-Tok Models on AWF100	67

List of Tables

2.1	Search spaces and their sizes	23
5.1	Hyperparameter space and final selections for DARTS optimization	60
5.2	Comparison of search strategies on undefended AWF traces	62
5.3	Comparison of search and train times per epoch for different strategies	63
5.4	Accuracy with and without early stopping on undefended AWF traces	65
5.5	Accuracy of NAS Models vs. Tik-Tok over 600 epochs	66
5.6	Accuracy for different search strategies and defenses with early stopping	68
5.7	DARTS_roneshot vs. Tik-Tok on increasing AWF traces per class	71
5.8	DARTS_roneshot and Tik-Tok accuracy for different time drifts	72

Chapter 1

Introduction

1.1 Overview

To protect their online privacy, users can employ encrypted tunnels, such as those established by low-latency anonymity networks such as Tor [DMS⁺04]. These tunnels enable users to create multi-hop encrypted pathways, concealing the contents and destination of their communications from potential eavesdroppers. However, Tor’s design choice to provide a low-latency (and thus, more enjoyable) browsing experience preserves most of the timing and volume properties of network packet exchanges observed when visiting a given website. These website-specific traffic properties, which persist over the encrypted tunnel [MD05], can be perceived as distinct traffic patterns that are characteristic of each website (i.e., fingerprints). The existence of these fingerprints, however, renders Tor susceptible to traffic analysis attacks, particularly website fingerprinting (WF) attacks [WCN⁺14, PLP⁺16, HD16, RPJ⁺18, SIJW18, RSM⁺20, BLKD19]. In these attacks, eavesdroppers exploit machine learning techniques to unveil the websites users visit by matching their traffic patterns to pre-recorded fingerprints.

Recent website fingerprinting research has transitioned from traditional machine learning techniques [PLP⁺16, HD16] to more precise attacks utilizing deep neural networks (DNNs) [RPJ⁺18, SIJW18, RSM⁺20, BLKD19]. However, the quest for improved DNN architectures driving WF attacks often relies on a trial-and-error approach. Researchers adapt or extend existing architectures from related domains, lacking a systematic understanding of factors influencing DNN performance.

While hyperparameter optimization methods [MCRG23, SLA12] aid in adjusting DNN parameters, the absence of a systematic approach impedes researchers from comprehend-

ing the factors contributing to performance enhancements across different DNN topologies. Without this understanding, decisions about architectural elements become heuristic or left to “pure chance.” Additionally, relying on existing DNN architectures for developing improved WF attacks may lead researchers to overlook unconventional or innovative architectural choices, missing potential breakthroughs.

Traffic analysis [DAAW22, HSFM21], particularly website fingerprinting [Gu22], has recently adopted automated machine learning (AutoML) [FKE⁺15, HKV19]. AutoML streamlines the traditional machine learning workflow by automating feature engineering, model selection, hyperparameter tuning, and model evaluation of classical machine learning models through various optimization techniques. Although AutoML’s success in finding highly effective models has led to wide application in various fields, traditional classifiers generated by AutoML have been less effective than deep learning models for WF [Gu22]. While many methods exist to optimize a deep neural network’s (DNN) hyperparameters, they do not address the challenge of finding the best architecture for specific tasks [MCRG23, SLA12].

To tackle the need for automated deep learning, the machine learning community has developed neural architecture search (NAS) [EMH19, RXC⁺21], which automates DNN architecture design by combining various operations and building blocks from a predefined search space. Our previous research utilized the AutoKeras NAS tool [JCSH23] to enhance WF attacks on Tor [SANM⁺23]. This preliminary study demonstrated that the generated DNNs achieved comparable accuracy to the Tik-Tok attack [RSM⁺20] on undefended Tor traffic and higher accuracy on traffic defended by random padding schemes [GW20]. However, the study was limited by a constrained exploration budget, a single search algorithm, and a single dataset.

This work aims to advance our understanding on the usefulness of NAS for WF, by seeking to validate the following thesis statement:

DNN architectures specifically searched and optimized on WF datasets will outperform existing DNN-based WF attacks that directly borrow or adapt DNN architectures from related domains.

In this thesis, we began by addressing some of the limitations identified in our preliminary study [SANM⁺23]. We began by evaluating the effectiveness of various search strategies while acknowledging the difficulties in implementing NAS algorithms and reusing existing code bases for a set of downstream tasks they were not originally developed for. To enable side-by-side comparisons of different NAS approaches, we utilized the capabilities of Microsoft’s Neural Network Intelligence (NNI) [Mic21] library. Although NNI is specifically

designed for tasks in computer vision, our contribution, FAWNS, adapts and integrates its flexible programming API to address website fingerprinting and traffic analysis.

Using FAWNS, we conducted comprehensive explorations of a NAS search space with various search strategies to evaluate FAWNS’ effectiveness in generating DNNs that are not only effective at fingerprinting websites but also robust and generalizable. The results of our evaluation show that FAWNS-generated DNNs achieve accuracy comparable to the Tik-Tok attack on undefended Tor traffic and significantly higher accuracy on defended Tor traffic. Specifically, FAWNS-generated models result in about a 26% accuracy increase on traces protected by the FRONT [GW20] defense and about a 12% increase with WTF-PAD [JAA+14] defended traces. Additionally, these models generalize as well as Tik-Tok on unseen data, indicating a substantial potential for NAS techniques to improve the effectiveness of WF.

1.2 Contributions

This thesis advances the state of the art in website fingerprinting by developing and evaluating the FAWNS framework, a tool for discovering and optimizing neural network architectures through NAS. In particular, this thesis makes the following contributions:

1. *Design and implementation of FAWNS.* By integrating with the NNI library, FAWNS creates a unified architecture for building and applying NAS algorithms to website fingerprinting. This integration simplifies the technical complexities of NAS, making it more accessible to researchers. Moreover, the framework’s modular and adaptable nature allows for easy extensions to other traffic analysis tasks,
2. *Systematic hyperparameter search for effective DNN exploration.* We conduct a systematic search to identify the most effective hyperparameters for exploring DNNs for WF. Using these parameters, we compare different search strategies within the FAWNS framework on undefended datasets to highlight their impact on the effectiveness of discovered architectures.
3. *Robustness towards defenses and concept drift.* We evaluate the performance of FAWNS-generated DNNs on defended traffic and assess their generalizability by comparing their performance across previously unseen data. Additionally, we study their robustness to concept drift, examining how these models maintain accuracy over time as the statistical properties of the traffic data evolve.

1.3 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 provides background on Tor, WF attacks and defenses, and NAS, along with a literature review. Chapter 3 introduces the FAWNS framework and outlines key aspects of its design, while Chapter 4 delves into the technical details of its implementation. Chapter 5 details the evaluation methodology and presents the experimental evaluation results. Chapter 6 concludes the thesis by highlighting the main findings and suggesting directions for future research.

Chapter 2

Background and Related Work

Despite Tor’s robust design, it remains vulnerable to website fingerprinting attacks. These attacks involve an adversary analyzing patterns in the encrypted traffic to infer the websites a user visits, thus compromising the privacy Tor aims to ensure. Over the years, researchers have developed various WF defenses to mitigate these attacks, aiming to make the privacy guarantees of Tor more resilient.

Parallel to these developments, the field of machine learning has seen significant advancements, particularly in NAS. NAS automates the design of neural network architectures, optimizing them for specific tasks, and has shown promise in various domains due to its ability to find highly effective architectures. Recently, the application of NAS to website fingerprinting has emerged as a novel research avenue [SANM⁺23].

In this section, we discuss the pertinent literature on Tor, WF attacks and defenses, and the emerging application of NAS in this domain.

2.1 Tor

Tor, short for *The Onion Router*, is a privacy-focused network that enables users to enhance their online anonymity by routing internet traffic through a series of volunteer-operated servers. The underlying principle of Tor is rooted in onion routing, where data undergoes encryption in layers, akin to the layers of an onion, and is subsequently relayed through various nodes within the Tor network. In this process, each node peels off one layer of encryption, revealing instructions for the subsequent hop. This cycle continues until the data reaches its final destination.

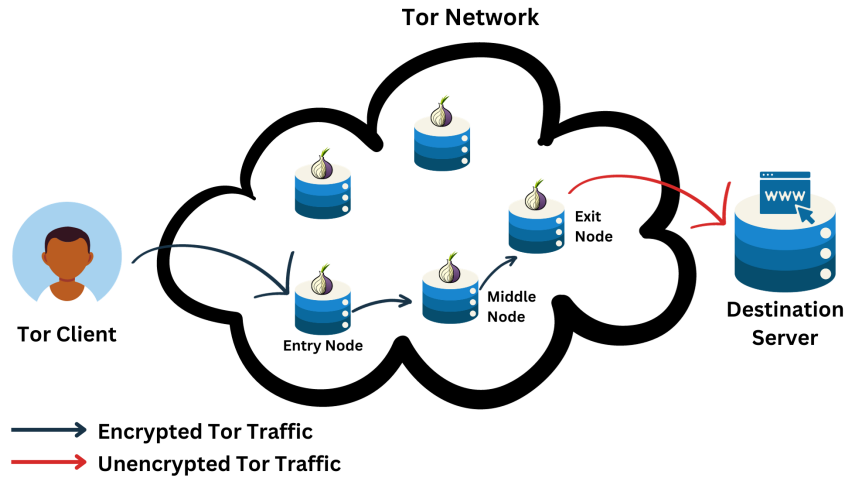


Figure 2.1: A typical Tor circuit.

The Tor network comprises a set of nodes, which are servers operated by volunteers participating in the relaying of encrypted traffic. There are three distinct types of nodes, as explained below.

1. *Entry Node (Guard Node)*: When a user initiates a connection to the Tor network, their traffic enters through an entry node. This node knows the user's IP address but does not know the ultimate destination of the traffic.
2. *Middle Node*: From the entry node, the traffic is forwarded to a middle node. Each middle node in the circuit only knows the node it received the data from and the one it is sending it to.
3. *Exit Node*: The final layer of encryption is removed at the exit node, and the traffic is sent to its destination. The exit node knows the final destination of the traffic but does not know the source.

To establish a connection through Tor, a user's device constructs a circuit comprising three nodes selected under certain restrictions. During the nascent stages of Tor, onion routing systems operated on a model of random selection of voluntary nodes within the network. However, the rapid growth in Tor's user base necessitated a more refined strategy for traffic management. Contemporary Tor architecture adopts load balancing, giving preference to nodes based on their perceived bandwidth capacity [JJH⁺17]. The client then forms a Tor circuit involving cryptographic key exchange and establishing networking sessions over the chosen nodes and routes the user's traffic over this circuit during the

session. The established Tor circuit is utilized for a specific browsing session or connection for approximately ten minutes before transitioning to a fresh circuit.

This triad of nodes (entry, middle, exit) enhances the difficulty for an observer to correlate incoming and outgoing traffic within the Tor network. With a minimum of three hops, no individual node possesses sufficient information to link the source and destination of a communication session.

2.2 Website Fingerprinting

Website fingerprinting (WF) is often seen as a classification problem, where an attacker seeks to identify if a client is visiting one of the web pages in a given set. In a standard WF attack targeting Tor, the adversary intercepts the communication between the user and the entry node of the Tor circuit. The preparation for the attack involves the adversary repeatedly visiting target websites, gathering network traffic traces, and extracting attributes that characterize these traces to construct a fingerprint database. Subsequently, the adversary constructs a model for website prediction. When attempting to identify the visited website of a target user accessing it through Tor, the adversary extracts the fingerprint and employs the pre-trained model for classification (see Figure 2.2).

In the context of WF research, evaluations of attacks often adopt the *closed-world* assumption, where users are restricted to visiting only those pages the adversary has trained on, i.e., *monitored* web pages. However, this assumption has been criticized as unrealistic due to the vastness of the Web, leading to the consideration of the more practical *open-world* scenario [JAA⁺14, Per13]. In the open world, users can access pages not included in the adversary’s training set, i.e., *unmonitored* web pages. While the closed world is still valuable for comparing existing attacks and defenses, this thesis comprehensively evaluates WF attacks in the closed-world scenario, leaving the open-world scenario to be addressed in future work.

2.2.1 Attacks

Attacks on simple privacy-enhancing technologies

Website fingerprinting attacks have been around since as early as 1998 as traffic analysis attacks. Over time, these attacks have demonstrated the ability to classify web-browsing traffic under progressively less stringent assumptions, underscoring the threat WF poses

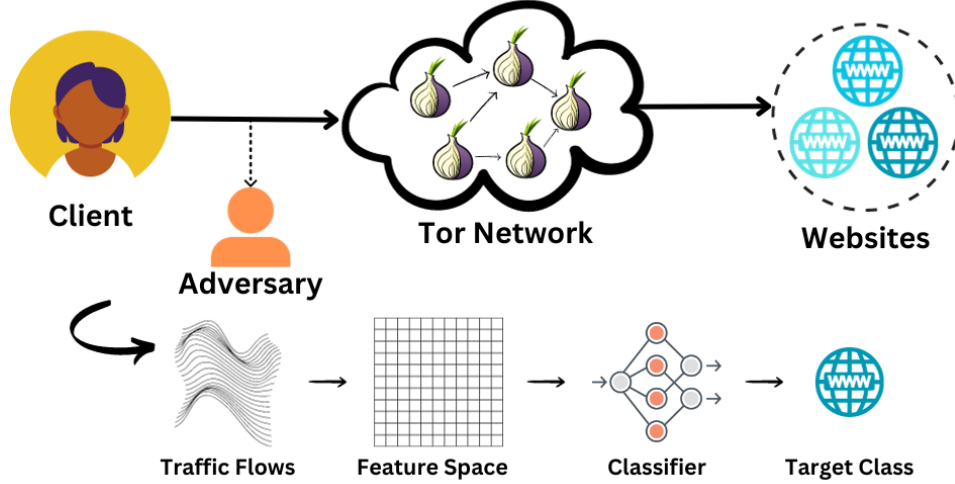


Figure 2.2: Overview of a website fingerprinting attack setup.

to the privacy of web-browsing activities. In this section, we outline various WF attacks and elucidate the classification methodology employed in each classifier.

Some of the initial WF attacks operated under the assumption that web resource lengths were disclosed to the attacker, since non-persistent HTTP/1.0 was in use. However, contemporary privacy-enhancing technologies, HTTP 1.1, VPNs, SSH tunneling do not leak resource lengths but expose packet lengths. As a result, research moved on to using deterministic packet lengths to fingerprint websites. However, as we shall see later, because of Tor’s fixed-sized cells, packet length-based attacks do not translate.

Given these criteria, we categorize attacks into two groups: those not intended for Tor, utilizing packet lengths, and those designed for Tor, which do not rely on packet lengths.

Packet length-based WF attacks. Liberatore and Levine [LL06] presented two packet-length related attacks, showcasing the potential for web page classification based on distinctive packet lengths. The first attack employed Jaccard’s coefficient as a similarity

measure, discerning comparable packet sequences through a substantial number of shared unique packet lengths. The second attack used a Naïve Bayes classifier and the classification of sequences was determined by analyzing packet lengths and their frequencies. Their method assumed independence between lengths and frequencies and adopted a statistical approach with Kernel Density Estimators. The size of the bins in these estimators was determined by the AMISE metric.¹

Herrmann et al. [HWF09] presented an enhanced version of Liberatore’s Naïve Bayes method by utilizing the Multinomial Naïve Bayes classifier. They incorporated sophisticated transformations from data mining such as term frequency (TF), inverse document frequency (IDF), and cosine normalization (CN) to emphasize less frequent lengths, reduce the impact of common lengths, and use relative frequencies for better performance.

Lu et al. [LCC10] devised an attack in response to the traffic morphing defense (see Section 2.2.2), specialized in distinguishing morphed packet sequences by focusing on the order of packet transmission. Their attack depends on the observation that the order of non-MTU packets (those not reaching the maximum transmission unit size) tends to remain consistent across packet sequences originating from the same web page. Levenshtein distance² was used to measure the similarity by calculating the number of insertions, deletions, and substitutions of packet lengths needed to transform one input packet sequence into another.

WF Attacks on Tor

The attacks outlined in the preceding section face challenges in being applied to Tor due to Tor’s fixed-length padding, where all cells are consistently 514 bytes long. To fingerprint Tor traffic, researchers needed to devise fingerprintable traffic characteristics beyond packet lengths. Initial WF attacks employed manually crafted features for training classical machine learning classifiers to fingerprint website visits over Tor connections [PNZE11, DCRS12, CZJJ12, WG13, WCN⁺14, PLP⁺16, HD16]. In recent times, WF research has witnessed a transition towards incorporating deep learning, employing automated feature extraction in its pipeline [RPJ⁺18, SIJW18, BLKD19, RSM⁺20]. We therefore

¹AMISE (Asymptotic Mean Integrated Squared Error), particularly in kernel density estimation, is used to measure the accuracy of an estimator. It provides a way to evaluate the performance of an estimator by considering the trade-off between bias and variance. A lower AMISE indicates a more accurate estimator.

²The Levenshtein distance is a metric for measuring the similarity between two sequences. It quantifies the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one sequence into the other.

classify WF attacks on Tor based on the feature extraction techniques they utilize: manual or automated.

WF attacks with manual feature extraction. Panchenko et al. [PNZE11] introduced an attack specifically aimed at web browsing clients utilizing anonymity technologies like JAP (Java Anonymous Proxy) and Tor. Unlike attacks that used unique packet lengths, which were ineffective on these anonymity networks, this attack by Panchenko et al., that utilized a support vector machine (SVM) with an RBF (Radial Basis Function) kernel, employed a variety of other features based on expert knowledge of web page functioning. They used markers to indicate direction changes in traffic flow and consequently obtained size markers by summing up packet sizes of the same direction. They counted the size of incoming packets between the request for the HTML document and subsequent requests for embedded objects, and used these HTML sizes as a discriminative feature. The attack also considered total transmitted bytes, percentage of incoming packets, and other aggregates as features. Their approach succeeded against JAP and Tor, achieving accuracy comparable to attacks on clients not using these anonymity technologies, marking the first successful attempt to do so.

Dyer et al. [DCRS12] introduced a variable n-gram naive bayes classifier on three features: the total time, the total length of all packets in each direction, and burst lengths. In their comparison with the SVM classifier used by Panchenko et al. across various scenarios, Dyer et al. did not observe their attack to be an improvement in any scenario. They attribute this to the fact that Panchenko’s attack employs a more intricate classifier, SVM, along with more detailed features. Literature suggests that the subpar performance of the n-gram classifier could be because of the inclusion of total time considering that the attacker cannot predict the client’s network conditions accurately [Wan16].

Cai et al. [CZJJ12] improved the accuracy of WF on Tor by introducing a new measure called the *optimal string alignment distance* (OSAD), which calculates the similarity between two packet sequences. A lower OSAD score indicates greater similarity. OSAD considers transpositions (swapping positions of items) in addition to other operations, making it a more comprehensive measure. This OSAD distance was used for kernel distance in their SVM, replacing the kernel distance typically used such as the RBF distance in Panchenko’s SVM. Noting that the authors rounded up each TCP packet length to the nearest multiple of 600 to approximate the count of transmitted Tor cells is useful. They were successful in attacking and defeating several published defenses, including Tor’s randomized pipelining defense [Per11] that enabled HTTP pipelining, randomized the pipeline size as well as the order of requests in order to impede the efficacy of Panchenko et al.’s SVM attack.

Wang and Goldberg [WG13] enhanced the accuracy of the previous attack on Tor by making several adjustments to OSAD. Their modified distance replaced the kernel in their SVM. Instead of using TCP packet sequences like Cai et al., they directly extracted Tor cell sequences from the captured packets. This was done by reconstructing TCP streams and parsing the SSL records within them. They also identify and remove SENDME cells.³

Wang et al. [WCN+14] devised the k -NN attack to address the open-world scenario. They employed the k-Nearest Neighbors (k-NN) classifier with about 4200 traffic features including total bandwidth, elapsed time, and packet characteristics. The process involved training with a set of packet sequences, using a large feature set to describe similarity, and employing a weight learning process called *weight learning by locally collapsing classes* (WLLCC). The k-NN attack is advantageous for its short training and testing times, suitability for multi-modal classes, and the flexibility to adjust parameters for a balance between true positive and false positive rates.

CUMUL [PLP+16] utilizes the cumulative sum of packet lengths to generate features for SVM. In this approach, for each packet in a network traffic instance, the length of outgoing packets is added, and the length of incoming packets is subtracted. For example, if a sequence of packets has lengths $[10, -5, 15, -8]$, the cumulative sum at each step would be $[10, 5, 20, 12]$. To meet the requirements of the RBF kernel used in SVM, which expects feature vectors to have the same dimension, CUMUL interpolates 100 points from the cumulative sums, ensuring consistent dimensions across different instances. To provide additional context, CUMUL incorporates the total number of incoming and outgoing packets as well as the total number of bytes into the feature vector. The success rate of the attack ranged from 90% to 93% across 100 websites.

Hayes and Danezis [HD16] introduced the k -fingerprinting (k-FP) attack, employing random forests (RF) – ensembles of randomized decision trees for enhanced generalization. Their feature set, which includes novel timing features like packets per second, comprises 175 features. Instead of employing direct classification, they leverage random forests to derive a fingerprint for each traffic instance. Subsequently, the extracted fingerprints undergo classification using a k-NN classifier using Hamming distance. The outcomes showcase the efficacy of this attack, demonstrating comparable accuracy scores to CUMUL.

WF attacks with automated feature extraction. Abe and Goto employed a stacked denoising autoencoder (SDAE), a neural network aiming to generate a condensed representation of its input, for automated feature engineering [AG16]. However, their model

³Tor utilizes the SENDME control cell for flow control [DMS+04]. These cells offer no additional information for fingerprinting and should be excluded, similar to the removal of ACK packets at the TCP/IP level.

exhibited inferior performance compared to Wang et al.’s k-NN with manually crafted features. It is reasonable to attribute the diminished performance to their insufficient quantity of training data for a deep neural network [RPJ⁺18].

Rimmer et al. examined the application of SDAEs, recurrent neural networks (RNNs), and convolutional neural networks (CNNs) to WF [RPJ⁺18]. The findings illustrated that models designed for automated feature extraction may showcase slightly superior performance compared to current approaches, especially when trained on extensive datasets. While they captured the timing information, direction, and size of the TCP packet, their ultimate representation of the traffic trace is a sequence of cells. Each cell is encoded as 1 when transmitted from the client to the website and as -1 when captured in the opposite direction. The resulting sequence is referred to as a *directional trace representation*.

Sirinam et al. introduced the *deep fingerprinting* (DF) attack in 2018 [SIJW18]. DF utilizes a CNN architecture inspired by Simonyan et al.’s VGG model [SZ15]. In both closed- and open-world scenarios, DF demonstrated advancements over previous approaches. When contrasting their findings with Wang and Goldberg’s Walkie-Talkie defense [WG17] (see Section 2.2.2), Sirinam et al. introduce the concept of Top-2 accuracy, the first paper to do so, that considers the top 2 probability values (and not only the highest probability (Top-1 prediction)). They accomplished a Top-2 accuracy of 98.44% against W-T, suggesting that DF adeptly identifies both the real site and the decoy site and, predictably, has to make a random choice between them.

Var-CNN [BLKD19] employs a ResNet (Residual Neural Network) structure, renowned for its proficiency in managing intricate tasks and enhancing training efficiency in deep learning models. The architecture integrates dilated causal convolutions, a type of convolutional layer that enables the network to grasp dependencies across a broader range, enhancing its capability to discern long-range patterns. In both scenarios, whether in open-world situations with abundant data or low-data, Var-CNN surpassed previous methodologies by achieving higher true positive rates (TPR) and lower false positive rates (FPR).

Rahman et al. present the *Tik-Tok* attack that leverages the CNN structure similar to DF [RSM⁺20]. However, in contrast to DF, its input involves the combination of direction and raw time, forming a new trace representation known as *directional-timing*, which enhances the attack’s effectiveness.

While state-of-the-art WF attacks have shown effectiveness even against defended Tor traffic, they face significant limitations in accurately identifying websites during multi-tab browsing sessions [JAA⁺14]. Deng et al. [DYL⁺23] address multi-tab WF by framing it as a multi-label classification problem and utilizing transformers for the solution. Jin et

al. [JLLS23] tackle multi-tab WF by using transformer architecture to extract WF features from different positions in the multi-tab traces more effectively.

Given that Tik-Tok is a widely adopted benchmark in WF research, the focus of this thesis is on utilizing NAS to generate convolutional neural networks, leaving the automatic discovery of transformer models for future work (Section 6).

2.2.2 Defenses

Website fingerprinting defenses play an important role in upholding the intended privacy, anonymity and censorship-resistant properties of low-latency anonymity systems in the face of increasingly sophisticated traffic analysis attacks. They do so by obfuscating the real characteristics of a website access trace. We now provide an overview of WF defenses.

Padding-based defenses. These strategies obscure timing patterns and packet transmission burst⁴ behavior by employing various approaches that depend on sending packets at consistent rates. Below, we describe prominent defenses which adhere to this strategy.

Cai et al.’s *BuFLO* (for Buffered Fixed-Length Obfuscator) defense [CZJJ12] involves packet padding, where all packets are padded to 1,500 bytes. Packet scheduling occurs at regular intervals. Additionally, BuFLO performs sequence padding only if the total number of packets is below $L_{\max}$, aiming to balance collision prevention with bandwidth and time overhead considerations. However, selecting an appropriate value for $L_{\max}$ remains a challenge, as increasing it can collide with more web pages but also raises overhead.

While BuFLO offers robust security, it comes with a high computational cost. In response, Cai et al. introduced *Congestion-Sensitive BuFLO* (CS-BuFLO) as an extension to BuFLO, incorporating various enhancements in terms of security and efficiency [CNJ14]. CS-BuFLO, akin to BuFLO, delivers fixed-size data chunks at semi-regular intervals. It also randomizes the timing of network writes while maintaining a target average inter-packet time ρ , occasionally updating it to align its bandwidth with the sender’s rate. The protocol utilizes TCP for congestion control, leveraging feedback from the TCP stack to minimize the amount of extraneous data it sends.

Tamaraw [CNW⁺14] by Cai et al. operates by sending fixed-size packets at fixed intervals, with a packet size of 750 bytes instead of the Maximum Transmission Unit (MTU). This size is chosen to cover most outgoing packets efficiently while minimizing unnecessary overhead. Outgoing and incoming traffic are treated differently in Tamaraw. Outgoing

⁴A burst contains consecutive packets in the same direction

traffic has a higher packet interval to reduce overhead, considering that outgoing traffic occurs less frequently. The packet intervals, denoted as ρ_{out} and ρ_{in} , are experimentally determined to optimize performance. Unlike BuFLO, Tamaraw always pads the number of packets sent in both directions to multiples of a padding parameter, denoted as L . This padding ensures that for each webpage, there is likely another webpage mapped to the same multiple of L .

In their 2006 publication, Shmatikov and Wang [SW06] introduced a defense mechanism, referred to as *adaptive padding*, in response to their own timing analysis attack. Adaptive padding functions by adding packets to obscure interpacket times from potential attackers. In this approach, intermediate mixes insert dummy packets into statistically improbable gaps in the packet flow utilizing two modes (burst and gap) to mimic natural traffic patterns, thereby disrupting timing fingerprints without introducing latency. The algorithm categorizes intervals into bins and employs statistical distributions to guide the insertion of dummy messages, preventing observers from discerning real communication patterns. Configurable and designed to minimize additional delays, the adaptive padding algorithm aims to enhance privacy by making message timing less predictable, thereby thwarting timing analysis attempts in network communication.

Originally conceived as a defense against end-to-end timing analysis, Juarez et al. introduced *WTF-PAD* (for Website Traffic Fingerprinting Protection with Adaptive Defense), a system designed to implement adaptive padding for WF defense in Tor [JAA⁺14]. This defense employs packet time sampling techniques to insert dummy packets during intervals between real packets, ensuring that actual traffic remains unaffected by any delays. WTF-PAD has demonstrated effectiveness against contemporary attacks with moderate bandwidth overheads compared to BuFLO-style defenses. Notably, WTF-PAD avoids introducing latency overhead as it does not delay packets.

RegulaTor [HH22] by Holland and Hopper regulates the size and shape of packet sequences in traffic to obfuscate potentially revealing features. The process involves initiating packet transmission at a predefined rate to mitigate information leakage about traffic volume and duration. Subsequently, RegulaTor reduces packet sending based on a specified decay rate parameter, shaping the size of the sequence. In the event of a scheduled packet without available data, a dummy packet is sent. Despite the burstiness of web-browsing traffic, this download padding approach incurs limited overhead.

Morphing-based defenses. *Traffic morphing* [WCM09] by Wright et al. involves randomly adding extra data to unique packet lengths. This modification is designed to make these lengths resemble those from a packet length distribution associated with a different web page. Analogous to how adaptive padding addresses interpacket times, morphing fo-

cuses on unique packet lengths. It is imperative to note that traffic morphing exclusively alters the unique packet lengths within a sequence, leaving sequence length, packet order, and inter packet timing unaffected. This targeted approach makes Morphing particularly effective against attacks that rely on distinguishing unique packet lengths, such as those devised by Liberatore and Levine (see Section 2.2.1).

Rather than distributing the data budget uniformly across the trace, Gong and Wang’s *FRONT* (for Front Randomized Obfuscation of Network Traffic) [GW20] allocates most resources to obfuscate the trace front as the beginning of the trace tends to leak valuable features used in website fingerprinting. *FRONT* introduces dummy packets in a highly random manner. This randomness ensures that different traces of the same webpage appear distinct in total length, packet ordering, and packet directions. This approach is particularly effective against attacks that rely on trace-to-trace consistency.

Gong and Wang’s *GLUE* [GW20] inserts dummy packets between traces to create the illusion of continuous page visits without interruptions. This strategy compels the attacker to grapple with challenging packet splitting issues, a task that previous research indicates even the most sophisticated attacks struggle to accomplish successfully [MHO⁺23].

Supersequence-based defenses. These defenses are crafted to counter WF attacks that attempt to classify traffic sequences based on differences among them. They achieve this by focusing on the *shortest common supersequence* (SCS) within a collision or anonymity set. The SCS of two or more sequences is the briefest sequence that encompasses all given sequences as subsequences. A collision set comprises distinct packet sequences that, when processed by the defense or anonymization system, yield identical or similar outputs. In essence, these diverse sequences “collide”, generating indistinguishable sequences from the attacker’s perspective. *Glove* [NCJ14] by Nithyanand et al. and *Supersequence* [WCN⁺14] by Wang et al. are examples of defenses that use this approach. Nevertheless, deploying these defenses becomes challenging for websites with dynamically loaded content, as the generation of supersequences necessitates prior knowledge about the website’s content.

Wang and Goldberg’s *Walkie-Talkie* defense (W-T) [WG17] extends these previous supersequence defenses by pairing sites and shapes traffic to make the directional sequence for each site identical. This involves organizing traffic into bursts, and padding each burst to create a supersequence. However, incorporating Walkie-Talkie presents a challenge as it requires a database containing burst sequences of numerous webpages for effective shaping of traffic bursts, a requirement that has proven challenging to fulfill in practical scenarios [RSM⁺20].

Application-based defenses. In 2011, Tor released an experimental *randomized pipelining* defense [Per11] to enhance user privacy by reducing the information available for feature

extraction in website fingerprinting attacks in response to Panchenko et al.’s SVM attack (see Section 2.2.1). The defense involves enabling HTTP pipelining and introducing randomness in the pipeline size and request order. This aims to counteract attack methods that rely on serialized request/response behavior. By randomizing the pipeline, the defense disrupts features used by attackers, such as the number of packets, packet ordering, especially on sites that support pipelining.

Luo et al. introduced *HTTP Obfuscation* (HTTPOS) as a defense against certain attacks on web privacy [LZC⁺11]. Unlike other defenses, HTTPOS operates on the client side without requiring cooperation from external sources. It uses features in the HTTP protocol, TCP advertised windows, HTTP pipelining, and HTTP ranges to control the sizes of both outgoing and incoming packets. HTTPOS employs a method involving splitting incoming packets into two random-sized packets by choosing a random value within the original packet size and setting a range header to create packets of varying lengths. Cai et al.’s OSAD [CZJJ12] (see Section 2.2.1) defeated HTTPOS.

ALPaCA (short for Application Layer Padding Concerns Adversaries) [CHJ17] by Cherubin et al. is a server-side defense aimed at concealing distinctive features on the network level through webpage content padding. It achieves this by padding various webpage objects, such as images and HTML, to a desired size without causing any visual alterations perceptible to users. The defense introduces two variants, namely P-ALPaCA (Probabilistic) and D-ALPaCA (Deterministic). Both variants involve selecting an appropriate list of sizes for a page to undergo morphing and specifying the number and size of objects for the morphed page. The key distinction lies in their approaches to choosing the target sizes for morphing.

Cherubin et al. also devised *LLaMA* (Lightweight application-Layer Masquerading Add-on) [CHJ17], a client-side defense mechanism inspired by randomized pipelining that tries to protect against website fingerprinting attacks by randomly delaying outgoing HTTP requests while introducing dummy HTTP requests.

Traffic splitting-based defenses. By intercepting traffic traces at a compromised Tor entry node, which is deemed a susceptible point in the Tor attacker model, a passive eavesdropper can utilize the obtained metadata to expose the websites visited by a Tor user. Some defensive strategies leverage this scenario by splitting traffic among multiple Tor entry nodes, dispersing the payload from a single site visit across different circuit hops in a way that conceals its source.

Following this idea, Henri et al.’s *HyWF* [HGS⁺20] connects to two independent access points (e.g., cellular network and home WiFi) and directs traffic to a Tor bridge that merges it before reaching Tor. De la Cadena et al.’s *TrafficSliver* [DICMH⁺20] either splits

traffic over multiple entry relays and merges traffic at the middle relay or creates multiple circuits and requests different HTTP objects over the circuits. Towards this, it offers two versions: TrafficSliver-App, which operates on the unmodified Tor network by proxying HTTP requests over multiple Tor circuits, and TrafficSliver-Net, which modifies the Tor network to split TCP traffic over multiple entry nodes before merging it at the middle relay.

Decoy-based defenses. In *Decoy*, introduced by Panchenko et al. [PNZE11], users load a decoy webpage simultaneously with the actual one, aiming to confuse attackers. To protect user privacy, the client does not reveal sensitive pages, preventing their use as decoys. Decoy is considered a limited defense, as the decoy page from the open world may not be of interest to the attacker focused on monitoring specific pages.

Trace generation-based defenses. In 2014, researchers discovered that deep learning models are vulnerable to a type of attack called adversarial examples. These attacks try to cause misclassifications by adding small amounts of adversarial noise to the inputs of the model. Inspired by this, the *Mockingbird* defense [RIMW20] devised by Rahman et al. creates adversarial traces by gradually changing a source trace until it no longer belongs to the source class. Shan et al.’s *Dolos* [SBZZ21] disrupts WF deep learning classifiers by generating input-agnostic adversarial patches, directing the injection of dummy packets into traffic traces. Nasr et al. introduced *BLANKET* [NBH21], a technique capable of thwarting deep learning WF attacks by indiscriminately perturbing the features of live connections. Holland et al.’s *DeTorrent* [HCOH23] employs Generative Adversarial Networks (GANs) to create defense strategies resilient to adversarial retraining. The defense generator, trained to make real-time decisions, is implemented as a Long Short-Term Memory (LSTM) model, considering live traffic patterns.

A more recent defense, *Surakav* [GZZW22] by Gong et al., generates traffic burst volumes for a trace, where a burst is defined as a series of packets going in the same direction, by training a Wasserstein GAN to emulate real traffic burst patterns from various web pages. To optimize efficiency and minimize overhead, rather than employing fixed sending patterns, Surakav dynamically adjusts the patterns based on the size of buffered data during a loading process.

The discussion of WF defenses reveals key lessons: defending against WF attacks requires diverse, multifaceted solutions, including padding, morphing, decoy, and trace generation. Each approach tackles different aspects of traffic analysis, demonstrating that no single solution provides complete protection. Defenses like BuFLO and Tamaraw involve trade-offs between security and computational or bandwidth overhead, emphasizing the need for practical solutions. The advancement of sophisticated attacks, particularly those

using deep learning, calls for the continuous evolution of defenses. This arms race highlights the importance of ongoing research to stay ahead of emerging threats. For a more detailed analysis of WF defenses, we refer readers to a study by Mathews et al. [MHO⁺23].

2.2.3 Datasets

Over the years, different research groups have gathered Tor traffic datasets with varying characteristics to study and experiment with WF attacks. Here, we present the website fingerprinting datasets used in our experiments or mentioned in this thesis.

AWF (2017). This dataset, collected by Rimmer et al. [RPJ⁺18] to evaluate their deep learning approach to website fingerprinting, represents the largest WF dataset compiled to date. Their closed-world dataset encompasses 900 websites, with each website’s traffic traces generated by 2,500 visits. Additionally, their open-world dataset is constructed from 400,000 unknown websites and 200 monitored websites. The list of websites chosen for evaluation is derived from the Alexa Top Sites service, a widely-used source in most Tor research.

Their evaluation compares their deep learning approach with traditional methods across three distinct datasets. The closed-world dataset, based on the closed-world assumption, involved collecting up to 3,000 network traces for visits to the homepages of the top 1,200 Alexa-ranked websites. Data collection for these websites occurred in four iterations, with each iteration comprising 300 websites divided into 30 batches. Each batch conducted 100 network traces per website.

Network traces for each of the four iterations were gathered over approximately 14 days per group, starting from January 2017. Post-collection, invalid entries resulting from timeouts or browser crashes were filtered out. Websites with a significant number of invalid page visits were excluded from the dataset. Additionally, websites with identical content, as determined by similarity hashes of the HTML content and perceptual hashes of screenshots, were identified and excluded. Websites with no content, those denying requests from Tor, or those presenting a CAPTCHA for every visit were also filtered out.

The dataset was then balanced to ensure uniform distribution of instances across different sites, ensuring an equal number of traces for each site. Following this filtering process, their largest closed-world dataset comprises 900 websites, each with 2,500 valid network traces. Subsets of this dataset are designated as AWF100, AWF200, and AWF450, representing datasets with 100, 200, and 450 traces per class, respectively.

DS-19 (2019). For their experiments, Gong and Wang collected a new dataset (referred to as DS-19) between February and April 2019 [GW20]. The data collection process was conducted via command-line calls to Tor Browser. They systematically visited the home-pages of the top 100 Alexa-ranked websites 100 times each to serve as their monitored webpages. Additionally, 10,000 other webpages were collected as non-monitored webpages, with pages failing to load (such as those inaccessible through Tor) being filtered out. This collection process utilized a single machine connected to a university network. As all traces were gathered from an automated browser and not from real users, there are no ethical concerns associated with the dataset or the subsequent experiments.

BigEnough (2022). Mathews et al. [MHO⁺23] gathered the BigEnough dataset in 2022, comprising three variants tailored for the Tor Browser Bundle’s three safety configuration settings: Standard, Safer, and Safest. Each variant consists of both monitored and unmonitored samples. The monitored set encompasses 19,000 samples spanning 95 websites, with each website represented by 10 subpages. Conversely, the unmonitored set comprises 19,000 samples of index pages for non-monitored websites.

2.3 Neural Architecture Search

In recent years, deep learning has surged to prominence across diverse domains, including computer vision [KSH12, HZRS16, HLVDMW17], natural language processing [BCB15, VSP⁺17], and reinforcement learning [MKS⁺15, SHM⁺16]. One of the key drivers behind the widespread adoption of deep learning is the automation of the machine learning (ML) workflow. This process of simplifying the machine learning workflow through automated feature engineering, model selection, hyperparameter tuning, and model evaluation is called automated machine learning (AutoML). Within the realm of AutoML, neural architecture search (NAS) has emerged as a dedicated subfield, focused on automating the design of neural architectures tailored to specific tasks.

NAS shares a considerable overlap with hyperparameter optimization (HPO), which involves the automated optimization of hyperparameters in machine learning models. While NAS is sometimes considered a subset of HPO, focusing on architecture-related hyperparameters, the techniques employed for HPO and NAS can differ substantially [WSS⁺23]. In NAS, the primary focus is on optimizing the topology of the architecture, which is often represented by a directed acyclic graph (DAG) where nodes or edges are labeled with neural network operations. This emphasis on architecture topology distinguishes NAS from traditional hyperparameter optimization.

Initially concentrated on image classification problems such as CIFAR-10 and ImageNet, NAS has expanded into diverse domains as evidenced by its recent adoption in the domain of network traffic analysis [MAS⁺23]. Within the context of this study, our objective is to utilize NAS capabilities to expand its application scope, specifically targeting website fingerprinting as a challenge to be addressed.

Surveys on NAS [EMH19, WRP19, WSS⁺23] commonly outline three essential dimensions of NAS: *search space*, *search strategy*, and *performance estimation strategy*.

Search space. Refers to the collection of all architectures that a NAS algorithm is permitted to select. While the search space can theoretically be all-encompassing, incorporating domain knowledge during its design can simplify the search process. It is crucial to strike a balance, as excessive domain knowledge may introduce human bias, diminishing the likelihood of the NAS method discovering truly innovative architectures.

Search strategy. Outlines how the exploration of the search space is conducted. In essence, it is an optimization technique employed to identify high-performing architectures within the defined space. Two primary categories of search strategies exist: black-box optimization-based techniques, and one-shot techniques.

Performance estimation strategy. Encompasses methods used to swiftly predict the performance of neural architectures, thus avoiding the necessity of fully training each architecture. Integrating a performance estimation strategy like learning curve extrapolation significantly accelerates the search process, avoiding the need to fully train and evaluate selected architectures throughout the search.

Now, placing everything together, NAS can be succinctly defined as the task of identifying an architecture a from a search space $\mathcal{A}$, given a dataset $\mathcal{D}$, training pipeline $\mathcal{P}$, and a specified time or computation budget t [WSS⁺23], further illustrated in Figure 2.3. The aim is to achieve the highest validation accuracy within the designated constraints, encapsulating the overarching goal of NAS methodologies.

Having established the broad framework of NAS, we shall now delve deeper into the components that make this process effective. One of the most critical elements in NAS is the search space. The structure and design of the search space significantly impact the performance and efficiency of the resulting neural architectures. Therefore, understanding and defining the search space is the first step in setting up any NAS process.



Figure 2.3: NAS dimensions [EMH19]: A search strategy iteratively picks architectures from a predefined search space $\mathcal{A}$. The chosen architectures are assessed by a performance estimation strategy, providing performance estimates back to the search strategy.

2.3.1 Search Spaces

The unique nature of the architectural search space is a defining characteristic of Neural Architecture Search (NAS). This space serves as the initial phase in the setup of NAS, often tailored to specific tasks and heavily influenced by state-of-the-art manual architectures in corresponding application domains.

The design of the search space involves a crucial trade-off between human bias and search efficiency. A smaller search space with many handpicked decisions facilitates NAS algorithms in discovering high-performing architectures more easily. Conversely, larger search spaces with primitive building blocks may require a longer runtime, but they hold the potential for uncovering truly novel architectures [RMS⁺17].

Chain-structured search spaces. Among the various types of search spaces, chain-structured search spaces exhibit a straightforward architecture topology characterized by a sequential chain of operation layers as outlined in Figure 2.4. While these spaces are conceptually simple and efficient to design, they often rely on state-of-the-art manual designs, potentially limiting the discovery of highly novel architectures. Notably, chain-structured search spaces, such as ProxylessNAS [CZH19], often build upon existing architectures like MobileNetV2 [SHZ⁺18].

Cell-based search spaces. The cell-based search space, a predominant type of search space in NAS, takes inspiration from the repetitive patterns found in state-of-the-art human-designed CNNs. Rather than searching for an entire network architecture from scratch, this approach involves searching over small cells, which are then stacked in sequence to form the overall architecture as shown in Figure 2.4. This significantly reduces the complexity of search spaces and has become highly popular in recent years. The transferability of cell-based structures, demonstrated by their effectiveness across datasets of varying sizes, adds to their appeal [ZVSL18, LSY19].

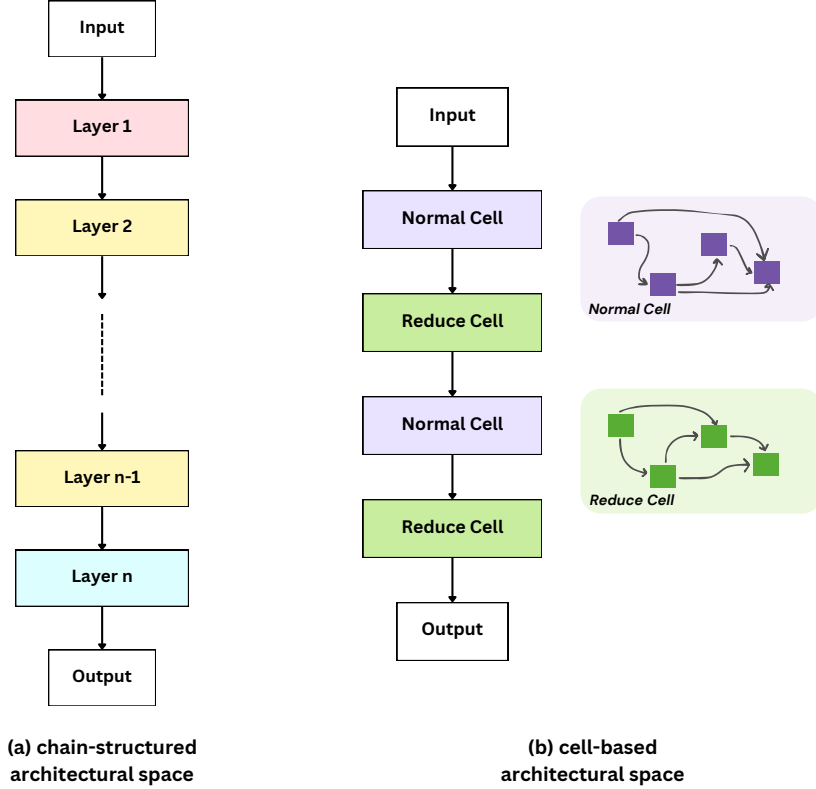


Figure 2.4: (a) Illustration of a chain-structured search space. Each node in the graphs represents a layer in a neural network, such as a convolutional or pooling layer. Different layer types are depicted using different colors. An edge between layers indicates that the subsequent layer receives the output of the preceding layer as its input; (b) Illustration of cell-based search spaces. An architecture is created by stacking multiple normal and reduce cells. Both normal and reduce cell architectures are searchable and represented as directed acyclic graphs, with operations assigned either to the edges or the nodes.

NASNet, proposed by Zoph et al. [ZVSL18], is an influential example of a cell-based search space with two types of cells: normal cells and reduction cells. The NASNet architecture is constructed by stacking these cells in sequence, where the normal cells retain spatial resolution and reduction cells reduce it. The NASNet cell features nodes representing operations and edges representing latent representations. Two of the most widely used cell-based search spaces are NAS-Bench-101 [YKC⁺19] and the DARTS search space [LSY19]. NAS-Bench-101 cells contain 7 nodes, each supporting 3 operations. DARTS, like NASNet,

employs two types of searchable cells with nodes representing latent representations and edges representing operations. Each cell in the DARTS search space has 8 edges offering eight choices of operations. The size comparison of these three discussed search spaces is detailed in Table 2.1.

Table 2.1: Search spaces and their sizes, sorted in decreasing order.

Search Space	Unique Architectures (#)
NASNet (2018)	10^{35}
DARTS (2018)	10^{18}
NAS-Bench-101 (2019)	423 624

Hierarchical search spaces. Hierarchical search spaces involve the design of motifs at different levels, where each higher-level motif is often depicted as a directed acyclic graph (DAG) of lower-level motifs [Liu18, CKM21] as depicted in Figure 2.5. In this context, a motif refers to a sub-pattern formed by multiple operations within an architecture. The first level encompasses a collection of basic operations, followed by a second level comprising various motifs connecting these primitive operations through a directed acyclic graph. The third level introduces motifs detailing the connections between second-level motifs, and this hierarchical pattern continues for subsequent levels. While most chain-structured and cell-based search spaces can be viewed as a hierarchical search space with a single searchable level, introducing two or more levels enables the exploration of more diverse and complex architectural designs. As hierarchical search spaces fall beyond the purview of this thesis, we refer the reader to White et al.’s survey [WSS⁺23] for an elaborate exploration of the attributes associated with this particular search space.

Understanding the intricacies of search spaces is only one aspect of the NAS process. Equally critical is how these spaces are navigated to discover optimal architectures, which we shall discuss in the next subsection.

2.3.2 Search Strategies

As discussed before, NAS search strategies are methods used to explore the search space of possible neural network architectures to find the optimal design. These strategies are crucial for determining how efficiently and effectively the NAS process can identify high-performing models. White et al. [WSS⁺23] categorize search strategies into two main groups: *black-box optimization techniques* and *one-shot techniques*.

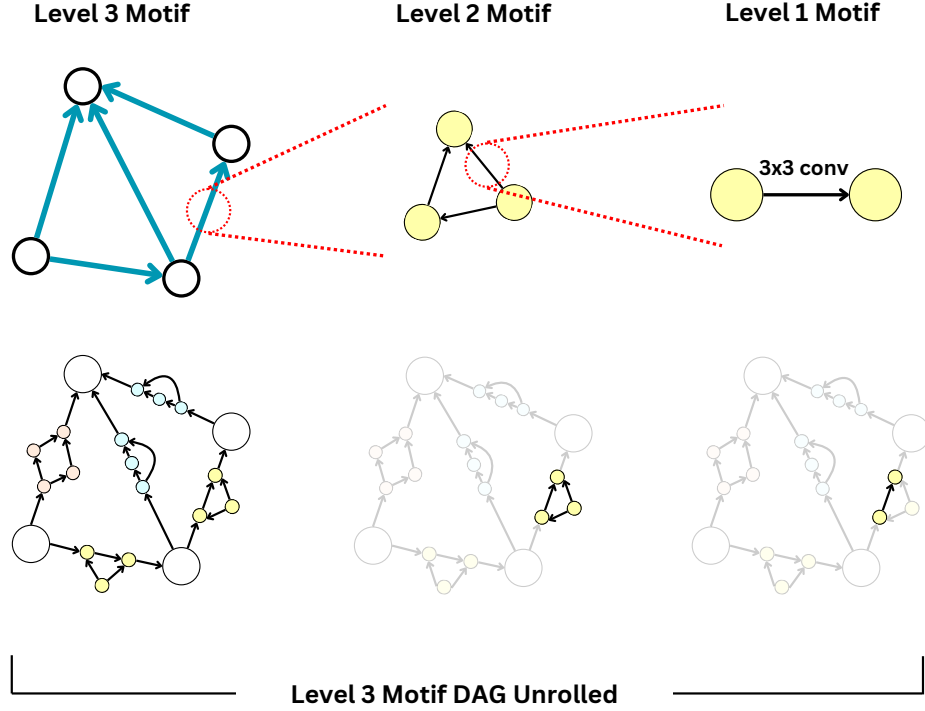


Figure 2.5: An example of a three-level hierarchical architecture representation proposed by Liu et al. [Liu18] includes the following levels: Level 1 involves choosing operation primitives. Level 2 focuses on selecting the topology across small sets of these operation primitives. Level 3 entails selecting the topology across the constructions derived from Level 2.

Black-box optimization techniques

Black-box optimization techniques are prevalent and extensively researched today due to their strong performance and user-friendly nature. Generally, these techniques tend to require more computational resources compared to one-shot techniques. This is attributed to the independent training of numerous architectures, unlike one-shot techniques that share weights across architectures. However, black-box optimization techniques offer several advantages over one-shot techniques, including but not limited to robustness, avoidance of

catastrophic failure modes, joint optimization with other hyperparameters, easier adaptability to new problems, datasets, or search spaces. Additionally, they are often conceptually simpler, making them more accessible for implementation and utilization [WSS⁺23].

Random search. When applied to NAS, random search involves the random selection of architectures from the search space, followed by full training, and outputs the architecture with the best validation accuracy. While random search performs well in highly engineered search spaces, its effectiveness diminishes on larger and diverse datasets [BLC⁺20, RLSL20]. Techniques like weight sharing [LT20], zero-cost proxies [AMDL21], or learning curve extrapolation [YWSH21] were developed to enhance the competitiveness of random search.

Reinforcement learning. When applied to NAS, reinforcement learning (RL) treats the generation of a neural architecture as the agent’s action. The reward is based on the performance estimate of the trained architecture on unseen data; i.e., validation accuracy is used as the reward signal. Different RL approaches use varying policies, with methods like ENAS employing strategies to speed up reward estimation [PGZ⁺18]. Although RL was initially prominent, it has been outperformed by evolutionary methods [RAHL19b] and Bayesian optimization in direct comparisons [YKC⁺19].

Evolutionary or genetic algorithms. When applied to NAS, genetic algorithms are used to refine a population of architectures iteratively. In each step, *parent* architectures are chosen based on validation accuracy and undergo mutations to generate new *children* architectures. These offspring are then trained and integrated into the population, replacing less-performing individuals. Initial architectures can be randomly selected [RAHL19b, SXZY19] or hand-picked for their high performance [FMM17]. The method for sampling parents varies; Real et al. [RMS⁺17, RAHL19a] and Liu et al. [Liu18] use tournament selection [GD91], while Elsken et al. [EMH18] opt for sampling from a multi-objective Pareto front based on inverse density. Other approaches include random sampling weighted by fitness [GLL18] or selecting the current best architecture(s) as parents [EMH17]. A notable evolutionary algorithm is *regularized evolution* by Real et al. [RAHL19b], adhering to a standard evolutionary approach but introducing the unique aspect of discarding the longest-standing architecture in each step, even if it has the best performance. This method outperformed random search and RL in comparisons, achieving state-of-the-art performance on ImageNet upon its 2019 release. To produce offspring, most approaches initialize child networks randomly. Elsken et al. [EMH18] introduced Lamarckian inheritance, passing knowledge (learned weights) from a parent network to its children through network morphisms.

Bayesian optimization. This optimization strategy provides an efficient approach for solving complex problems, relying on informed guesswork and strategic exploration to pro-

gressively identify optimal solutions. It consists of two key components: 1) *Surrogate model* to predict the unknown objective based on past observations, and 2) *Acquisition function* to strategically balance exploration and exploitation during the search. Initially, bayesian optimization for NAS used custom distance metrics for comparing architectures [SDS⁺14, KNS⁺18, NLYO21], enabling a Gaussian process-based surrogate within bayesian optimization. Challenges arose due to the intricate nature of NAS spaces, prompting two main strategies: encoding architectures before training models [BBBK11, HHLB11, SKFH16] and projecting architecture information into a simplified space [REC20, WREC22]. Some models use graph techniques for architectures without explicit encoding [MCY19, RWDO21, SPX⁺20]. The common approach for optimizing the acquisition function in NAS involves randomly mutating a small pool of the best-queried architectures, selecting the one(s) with the highest acquisition function value [KNS⁺18, MCY19, RWDO21, SPX⁺20]. Other methods include local search, evolutionary search, and random search [RWDO21, SPX⁺20, YKC⁺19].

Monte Carlo tree search. Originally devised for decision-making in games and simulations, Monte Carlo tree search (MCTS) systematically explores and evaluates potential moves to identify the most promising option. The process involves path selection in a decision tree from the current game state (selection), tree expansion by considering new moves (expansion), simulation of random playthroughs for outcome estimation (simulation or rollout), and updating node statistics based on simulation results (backpropagation). This iterative cycle constructs a tree structure, aiding the algorithm in making well-informed decisions. MCTS found its initial application in NAS by Negrinho and Gordon [NG17], who utilized a modular language to represent the search space and its hyperparameters, creating a flexible and extensible search space. Wistuba [Wis18] introduced a similar method employing two distinct Upper Confidence bounds applied to Trees (UCT) algorithms. Wang et al. [WZJ⁺19] adapted MCTS to cell-based search spaces using a state-action representation.

One-shot techniques

In early NAS research, the prevalent black-box optimization methods, as discussed earlier, incurred significant computational costs when applied without speedup techniques (see Section 2.3.3) due to their independent training of numerous architectures. One-shot techniques emerged as an alternative to mitigate the need for training each architecture from scratch, reducing computational burdens. They are now one of the most widely adopted approaches in NAS research.

One-shot approaches simultaneously train all architectures in the search space through a unified *one-shot* training session, employing either a hypernetwork or a supernet. A hypernetwork is a type of neural network that generates the weights for another neural network, which is referred to as the target network. The hypernetwork is trained to produce the parameters of multiple architectures within a predefined search space, effectively enabling the evaluation of various architectures without the need to train each one from scratch. We will not delve into it any further as it falls outside the purview of our study. A supernet (also known as a one-shot model), initially introduced by Saxena and Verbeek [SV16], acts as an over-parameterized architecture that includes all potential architectures as subnetworks. Supernets streamline the evaluation of individual architectures by inheriting weights from the corresponding subnet within the supernet.

Supernets achieve scalability and efficiency by ensuring that, despite a linear increase in candidate operations, training costs rise linearly, while the number of subnets in the supernet grows exponentially. This enables the training of an exponential number of architectures for a linear compute cost [WSS⁺23]. One-shot approaches make the crucial assumption that using the one-shot model to evaluate architectures results in rankings consistent with those obtained from independent training. The validity of this assumption depends on factors such as search space design, one-shot model training techniques, and the dataset, posing challenges in predicting its extent in a given case [YSJ⁺20, ZLJ⁺20].

A widely known example of one-shot training with supernets is Liu et al.’s DARTS (Differentiable ARchiTecture Search) [LSY19]. Instead of exploring a discrete set of potential architectures, DARTS transforms the search space into a continuous domain. This enables the optimization of architecture with respect to validation set performance using gradient descent. The data efficiency of gradient-based optimization, in contrast to inefficient black-box search methods, allows DARTS to achieve competitive performance while utilizing significantly fewer computational resources.

To summarize, NAS search strategies are divided into black-box optimization techniques and one-shot techniques. Black-box optimization approaches, such as reinforcement learning and evolutionary algorithms, treat architecture search as an optimization problem, but are often computationally intensive. One-shot techniques, like DARTS, reduce this burden by training a single over-parameterized model (supernet) encompassing all possible architectures in the search space, using gradient-based optimization for efficiency. These strategies are pivotal in balancing search efficiency and computational resources in the NAS process. We now turn to the third dimension of NAS: performance estimation strategies, which are essential for minimizing the need for fully training each candidate architecture during the evaluation phase.

2.3.3 Performance Estimation Strategies

Performance estimation strategies present the opportunity to significantly expedite NAS algorithms by circumventing the need for complete training of each architecture during the evaluation phase. White et al. [WSS⁺23] provide a formal definition for the performance estimator, denoted as f' , which is any function capable of predicting the accuracy or relative accuracy of architectures without the necessity of full training for a given search space A and an architecture $a \in A$, with $f(a)$ being the final validation accuracy achieved through a fixed training pipeline. In simpler terms, evaluating $f'(a)$ requires less time than assessing $f(a)$, and the ideal scenario is that $\{f'(a) \mid a \in A\}$ exhibits a high correlation or rank correlation with $\{f(a) \mid a \in A\}$.

Performance estimations can involve *proxy metrics*, serving as lower-fidelity approximations of full training outcomes. These proxies include reducing training times [ZVSL18, ZKFH18], training on data subsets [KFB⁺17], using lower-resolution images [CLH17], and fewer filters per layer and cells [ZVSL18, RAHL19a]. While these approximations reduce computational costs, they introduce bias by typically underestimating performance. This bias may not pose issues if the search strategy relies solely on architecture ranking, and the relative ranking remains stable.

An alternative method for estimating architecture performance involves learning curve extrapolation, where initial learning curves are extrapolated, and poorly predicted ones are terminated to expedite the architecture search. Learning curve extrapolation methods can also be integrated with a surrogate model. Here, the model receives both an encoding of the architecture and a partial learning curve as input, producing a prediction $f'(a)$ [BGRN17, KFB⁺17].

Another approach to accelerating performance estimation involves initializing the weights of new architectures based on weights from previously trained architectures. *Network morphisms*, introduced by Wei et al. [WWRC16], allow the modification of an architecture while preserving the represented function. This approach, adopted by Elsken et al. [EMH17], Cai et al. [CCZ⁺18, CYZ⁺18], and Jin et al. [JSH19], significantly reduces computational demands. However, strict network morphisms can only increase architecture size, potentially resulting in excessively complex structures. To address this, approximate network morphisms have been proposed, permitting both enlargement and reduction of architectures [EMH19].

One-Shot architecture search treats all architectures as distinct subgraphs of a supergraph known as the one-shot model. This approach involves sharing weights between architectures that have common edges in the supergraph [SV16, BLRW18, PGZ⁺18, BKZ⁺18,

[CZH19,XZLL19]. Only the weights of a single one-shot model need to be trained, and architectures, as subgraphs of the one-shot model, can be evaluated without separate training by inheriting trained weights. This accelerates architecture performance estimation since no training is needed. The one-shot model tends to introduce a significant bias by underestimating the actual performance of the best architectures. However, it enables ranking architectures, which is sufficient if the estimated performance strongly correlates with the actual performance, although current evidence remains inconclusive [BKZ⁺18,YSJ⁺20]. It should be duly noted that various one-shot NAS methods differ in how the one-shot model is trained.

2.4 NAS for WF

Building on AutoML’s success in traffic analysis and WF, our previous study [SANM⁺23] was the first to apply neural architecture search to the task of website fingerprinting. This study used AutoKeras [JCSH23], an open-source NAS system that utilizes Bayesian optimization to guide DNN morphism. AutoKeras focuses on deep learning tasks, distinguishing it from AutoML tools that target shallow neural network models [FKE⁺15,PLP⁺16]. Results from our exploration over the DS-19 dataset showed that DNNs created from scratch by AutoKeras achieved accuracy comparable to the Tik-Tok attack [RSM⁺20] on undefended Tor traffic. Moreover, AutoKeras-developed models outperformed Tik-Tok on defended Tor traffic, with an accuracy increase of 5-8% on traces protected by the FRONT defense [GW20].

Despite the promising results, this study had several limitations. It focused exclusively on AutoKeras, thus evaluating only a single exploration strategy for the applicability of NAS in the WF domain. Additionally, the experiments were constrained by a limited exploration budget of 100 trials and did not assess the generalizability of AutoKeras-generated DNNs or the potential of NAS tools with larger datasets.

To build upon our previous work and address its limitations, we developed a new neural architecture framework named FAWNS. This framework is designed to support various WF datasets, search spaces, and search strategies, providing a more comprehensive approach to neural architecture search in the domain of website fingerprinting. By leveraging FAWNS, we aim to facilitate a more thorough evaluation of different NAS techniques, enhancing the effectiveness and generalizability of the resulting DNN models.

The AutoKeras library, while user-friendly, offers limited customization and flexibility, especially in defining the search space and strategy. Users face restrictions in adding

new layers, operations, or connections, limiting the exploration of novel architectures for specific applications. This is due to AutoKeras providing predefined search spaces for common tasks like image and text classification. Additionally, AutoKeras employs a multi-trial approach for NAS, where each trial represents a separate architecture that is fully trained and evaluated independently. This method is time-consuming and computationally expensive, requiring the training of multiple models to completion.

Considering these limitations—restricted customization of the search space, the time-intensive nature of the multi-trial approach, and the high computational cost—we have opted to exclude AutoKeras from our evaluation in this thesis. By doing so, we focus on more flexible and efficient NAS methods that better suit our research needs. In the next chapter, we delve into the design of FAWNS, detailing its architecture and key components that make it an effective tool for WF research.

Chapter 3

The FAWNS Framework

FAWNS¹ is a neural architecture search framework engineered for the specific demands of WF, such as dealing with Tor traffic data, generalizing across different websites and user behaviors, and countering WF defenses. This framework is built using the PyTorch library in conjunction with Microsoft’s Neural Network Intelligence (NNI) [Mic21] for automating and optimizing the search for the most effective neural architectures.

In this chapter, we provide a comprehensive overview of the FAWNS framework, detailing the key modules (and interactions thereof) that deliver FAWNS’ core functionalities.

3.1 Extractor

This initial component of the FAWNS workflow manages input traffic traces in the format established by Wang et al. [WG13]. It extracts features from *website-trace* datasets such as AWF or DS-19, which consist of traces collected from visiting the homepages of various websites. It also handles *website-webpage-trace* datasets like BigEnough, where traces from different webpages within a website are gathered. Additionally, it incorporates high-fidelity simulators of defended Tor traffic, borrowed from previous research [VRB23, HH22], into its processes. When necessary, the **Extractor** can also simulate specified defenses on the undefended traces from one of the supported datasets. Moreover, this module can prepare datasets for both attack scenarios in website fingerprinting: open-world and closed-world.

¹FAWNS is derived from scrambling NASWF (Neural Architecture Search for Website Fingerprinting).

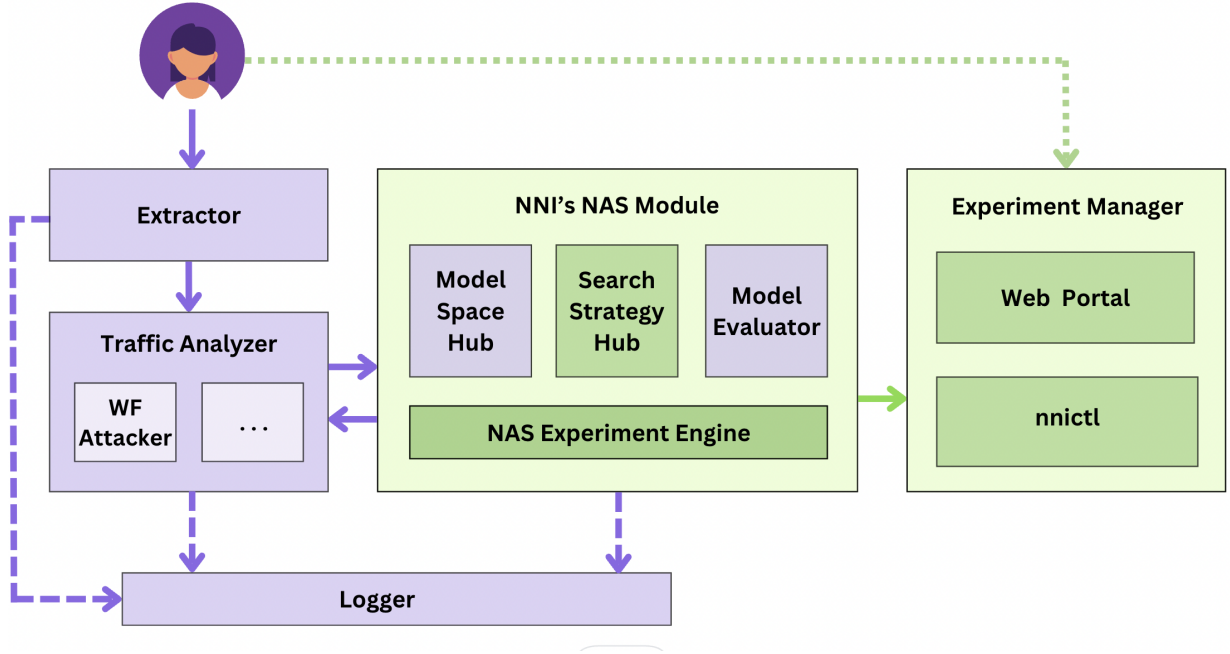


Figure 3.1: Overview of the FAWNS architecture. Modules colored lilac are either original to FAWNS or have been adapted with modifications. Modules colored green have been adopted without any changes. Lilac arrows represent FAWNS-specific integrations, and green arrows indicate integrations by NNI. Lilac dashed arrows show messages logged by the **Logger**. A green dotted arrow illustrates user interactions with the **Experiment Manager** for controlling NAS trials or visualizing experiments.

Essentially, the primary function of the **Extractor** is to derive the directional-timing features from the input traces to create datasets ready for further processing. After extracting these features, the **Extractor** applies standard preprocessing functions such as padding or truncating each trace to a uniform length and splitting datasets into smaller subsets. This data splitting feature can downsample large datasets in a stratified manner to ensure proportional representation of each class. This is crucial not only for preventing bias in the learning process, especially in cases of unbalanced datasets but also for significantly reducing the size of the dataset, whether balanced or not. Such reductions are beneficial as they save on computational power and training time, thereby enhancing the cost-effectiveness of the NAS process.

Ultimately, the extracted features are either loaded into memory for immediate use by subsequent modules or stored in files for later use. The modular design of the **Extractor**

allows it to support a wide range of WF datasets beyond AWF, DS-19, and BigEnough, and to include defense simulators beyond FRONT, WTF-PAD, Tamaraw, and RegulaTor that are currently supported for our experiments (see Section 5.5).

3.2 Traffic Analyzer and WF Attacker

Traffic Analyzer module stands at the core of FAWNS and handles input data from the **Extractor** or alternatively, loads already extracted and stored features and labels from input files, and applies preprocessing tailored for classification tasks such as WF before launching the NAS process. It performs operations like reshaping, data augmentation, and dataset division into training, validation, and test sets. The design of the **Traffic Analyzer** accommodates the typically one-dimensional traffic traces used in traffic analysis tasks by reshaping them for NAS applications, which are inherently designed for high-dimensional image processing. Its modular architecture allows for easy adaptation to new traffic analysis challenges and supports the integration of additional submodules as necessary. Currently, the **WF Attacker** is the only supported submodule, and we discuss its design below.

WF Attacker submodule

This submodule performs WF-specific preprocessing, prepares training, validation, and testing splits from preprocessed traffic features, and can handle both open-world and closed-world scenarios for website fingerprinting. It also oversees architecture exploration using NAS by initiating the model search process, training the optimal architecture identified by NAS, and gathering results throughout the process. These results, comprised of accuracy and loss indicators from the search, training, and testing phases, provide comprehensive evaluations of the website fingerprinting task.

Preprocessing WF directional-timing feature vectors for NAS. A critical preprocessing step for the one-dimensional WF directional-timing feature vectors is reshaping them for compatibility with NAS, given that NAS primitives are typically designed for image processing tasks. In state-of-the-art attacks such as Tik-Tok, a feature vector is initially represented as a simple sequence of Tor cells (in the directional-timing format) of length 5,000. The **WF Attacker** preprocessor transforms this one-dimensional vector of into a 5,041-element vector by appending trailing zeros, then converting it into a 71x71

matrix. Each cell in the matrix is then expanded into three channels to mimic the composition of an RGB image, resulting in inputs of the shape $[71, 71, 3]$. Each cell in the matrix is then expanded into three channels to mimic the composition of an RGB image, resulting in inputs with the shape $[71, 71, 3]$. In other words, the original $[71, 71]$ matrix is duplicated two more times, and the three $[71, 71]$ matrices are stacked together to form a 3-dimensional matrix with the shape $[71, 71, 3]$. The number 71 holds no inherent significance in the context of Tor cells or website fingerprinting; it was primarily chosen to create a square matrix close to the original 5,000 elements. However, this 71×71 2D matrix may still capture broader patterns in the data, even if the adjacency does not correspond to the original sequence, as evidenced by prior traffic classification research [WZZ⁺17, ACMP18]. NAS-generated architectures often feature multiple layers and varied operations that can jointly capture complex dependencies. By stacking multiple convolutional layers, pooling layers, and applying non-linear transformations, the network can integrate local and global temporal patterns. Moreover, as we shall discuss in Section 3.3, NAS search spaces apply depthwise concatenation to latent feature representations corresponding to intermediate nodes of NAS cells, consequently capturing relevant temporal dependencies before flattening them for a fully connected classifier. Nevertheless, we recognize that 1D networks are preferred and widely used to preserve temporal dependencies in sequential WF data, and we plan to adapt NNI’s search algorithms for these lower-dimensional inputs in the future.

Following the reshaping of vectors, this module generates a 9:1 training-testing split of the inputs in a stratified manner. The test split is reserved solely for making predictions with the final trained model.

Dataset downsampling considerations for NAS. NAS techniques, similar to other deep learning algorithms, are most effective with large datasets. For example, popular NAS methods are benchmarked on the CIFAR10 dataset, which contains 60,000 32×32 color images across 10 classes, and are evaluated for generalizability on the significantly larger subsets of the ImageNet dataset, which includes at least 1.4 million 224×224 images. In contrast, the complete AWF dataset contains 450,000 traffic traces distributed across 100 classes. Utilizing all these traces would necessitate extensive time and computational resources for searching and training models using NAS techniques. To optimize resources while conducting a broad range of experiments with FAWNS, we opted to use a downsampled version of the AWF100 dataset (10,000 traces across 100 classes). Even with downsampling, we can effectively evaluate the generated models’ performance by using some of the omitted data not included in the training setup. This allows us to test the models’ generalizability and assess their robustness on a broader dataset (see Section 5.6).

Enhancing data utilization with cross-validation. To address overfitting and maximize data usage in these data-limited (compared to NAS benchmarking datasets) scenarios, this module uses k-fold stratified cross-validation to further divide the training set into a 9:1 train-validation split k different times. These splits are then processed by the **NAS Experiment Engine**, which executes the selected exploration strategy to identify the optimal architecture that is fine-tuned on this train-validation split. This chosen architecture’s weights are subsequently reinitialized and trained on the same split to develop the final model, which is then used to make predictions on the reserved test split. The effectiveness of the model is determined based on the accuracy of these predictions in the closed-world scenario and on the precision, recall and precision-recall (PR) curves of these predictions for open-world scenarios.

Tracking and reporting across cross-validation folds. The **WF Attacker** is responsible for exporting the best model identified across all cross-validation folds, along with its training, testing, and timing statistics, to preserve the top-performing configurations and results. Additionally, it tracks and reports fold-level training and testing statistics for each model search-then-train cycle within each cross-validation fold. This includes detailed fold-specific data such as training outcomes, test results, and execution times for each fold. This systematic data collection enables a thorough post-training analysis, allowing for a detailed review of the model’s performance across different data subsets.

The **WF Attacker** integrates directly with **NNI’s NAS Module** to conduct both the architecture search and the training of the best architecture. The design of this integration is discussed in the following subsection.

3.3 NNI’s NAS Module

To effectively run NAS experiments, this module must configure and integrate the three NAS dimensions—search space, search strategy, and evaluation scheme—with the NAS experiment engine. Consequently, the implementation details are structured into four subsections: a) the **Model Space Hub**, b) the **Search Strategy Hub**, c) the **Model Evaluator**, and d) the **NAS Experiment Engine**. This organization systematically details the creation of search spaces, the use of NNI’s search strategies, the setup of an evaluator, and the configuration of NAS experiments.

3.3.1 Model Space Hub

The **Model Space Hub** comprises a repository which includes both predefined search spaces provided by NNI and custom-defined search spaces created by the user. To support the creation of these custom search spaces, NNI offers a diverse array of mutation primitives. We utilize some of these primitives to replicate the DARTS search space for our experiments with FAWNS during the evaluation. We chose the DARTS search space based on a study by Radosavovic et al. [RJX⁺19], which demonstrates that exploring the DARTS search space yields a higher number of well-performing models (with low validation errors) compared to other search spaces like ENAS, PNAS, AmoebaNet, and NASNet. Additionally, Radosavovic et al. show that for a fixed search algorithm, variations in design spaces alone result in significant performance differences. Therefore, in our experiments, we keep the DARTS design space fixed while varying the search algorithm. We will discuss the concept behind the DARTS search space next.

The DARTS search space. In line with the research by Zoph et al. [ZVSL18], Real et al. [RAHL19b], and Liu et al. [LZN⁺18], the DARTS exploration strategy searches for a computation cell that acts as the fundamental component of the final architecture. A cell is visualized as a directed acyclic graph composed of an ordered sequence of nodes, each representing a latent feature like a feature map in convolutional networks. The directed edges between these nodes carry out specific operations that modify the feature representation from one node to the next.

As shown in Figure 3.2, the input nodes for a convolutional cell k are typically the outputs from the two preceding layers, $k - 2$ and $k - 1$, which aids in capturing hierarchical feature information. The cell’s output is generated by applying a reduction operation, such as concatenation, to all intermediate nodes, n_0, n_1, n_2, n_3 , thus integrating the transformed features from all nodes into a single output representation. Each intermediate node’s representation is computed by aggregating the operations applied by all its preceding nodes, ensuring that each node integrates features and transformations from earlier in the sequence. More specifically, each node takes two previous nodes within the same cell (or the two cell inputs), applies an operator (e.g., convolution, or max-pooling) to each input, and sums the outputs of operators as the node’s output. A special zero operation is also included to indicate a lack of connection between two nodes. Therefore, learning the architecture of a cell reduces to learning the operations on the edges between its nodes.

Liu et al. designed the DARTS search space by incorporating four nodes per cell, with the cell’s output being the concatenation of all the nodes. This search space includes eight candidate operations listed below:

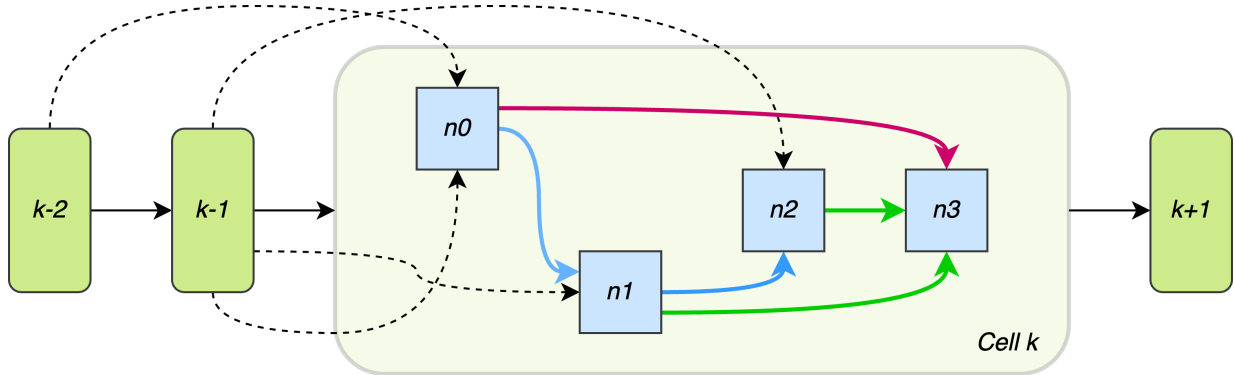


Figure 3.2: An overview of DARTS: Each cell takes as input the outputs from the two preceding cells. Each cell contains four nodes with candidate operations on the edges. The output of the cell is the depthwise concatenation of these four intermediate nodes. The differently colored arrows for the edges indicate three different operations selected from the list of candidate operations detailed in the subsection on the DARTS search space.

- **none**: Represents no operation.
- **max_pool_3x3**: A max pooling operation using a 3x3 filter.
- **avg_pool_3x3**: An average pooling operation with a 3x3 filter.
- **skip_connect**: A skip connection that directly transmits the input to the output.
- **sep_conv_3x3**: A separable convolution with a 3x3 filter.
- **sep_conv_5x5**: A separable convolution using a 5x5 filter.
- **dil_conv_3x3**: A dilated convolution with a 3x3 filter.
- **dil_conv_5x5**: A dilated convolution using a 5x5 filter.

DARTS and model architecture downscaling. A one-shot exploration technique such as DARTS integrates multiple architectural choices into a single trainable model by assigning weights to each architectural component. These weights are learned during the training process, allowing the model to evaluate many architectural configurations simultaneously. Due to this fact, one-shot search techniques like DARTS demand significantly more computational resources than traditional training methods, often resulting in slower training times and increased memory usage.

To manage the resource-intensive nature of one-shot strategies, a common practice is to downscale the network during the architecture exploration phase. This involves reducing the width, i.e., the number of filters in each cell, and the depth, i.e., the total number of stacked cells, which accelerates the training process while still capitalizing on the benefits

of architecture search. Downscaling ensures that the NAS process remains computationally manageable without substantially compromising the performance of the resulting model.

To perform a downscaling amenable to DARTS, one can leverage the fact that it is possible to design a single convolutional model architecture by composing multiple small modules, instead of designing the architecture all at once [ZVSL18]. This method draws from engineering efforts targeted at CNNs, which reveal that effective architectures typically contain repeated motifs. These motifs consist of combinations of convolutional filter banks and nonlinearities, similar to the repeated modules found in the Inception [SLJ+15] and ResNet models [HZRS16]. These insights suggest that an exploration strategy could successfully predict a generic convolutional cell defined by these motifs and that this cell could then be stacked sequentially (eventually configured with different weights) to manage inputs with varying spatial dimensions and filter depths. Cells of NAS search spaces, therefore, are categorized into two distinct types based on their functionalities, which we detail below.

- *Normal cells* process input feature maps without changing their dimensions, maintaining the spatial integrity of features across the network. More technically, convolutions and poolings in this block have a stride of 1.
- *Reduction cells* reduce the dimensions of input feature maps, typically by halving both the width and height, which helps in aggregating broader contextual information and reducing computational demands in deeper layers. More technically, convolutions and poolings in this block have a stride ≥ 2 .

The design of the DARTS search space, with its candidate operations for the edges between nodes of a cell, allows search strategies to select the best operation for these edges during architecture training. This process results in the identification of optimal normal and reduction cell architectures. Once the optimal normal and reduction cells are selected, they are assembled into a final network architecture as depicted in Figure 3.3. This assembly typically involves scaling up the width and number of cells. Increasing the width of the final model enhances its capacity to capture more complex features and allows the network to generalize better across larger and more diverse datasets. Moreover, increasing the number of cells deepens the network, allowing for more levels of feature extraction and transformation.

In NAS, the process of determining the architectures for normal and reduction cells involves employing a predefined strategy to navigate through a vast array of possible designs in the search space. The goal is to select the most effective architectures based on performance metrics, such as validation accuracy and generalization ability. These exploration strategies are part of the NAS module’s **Search Strategy Hub**, which we cover next.

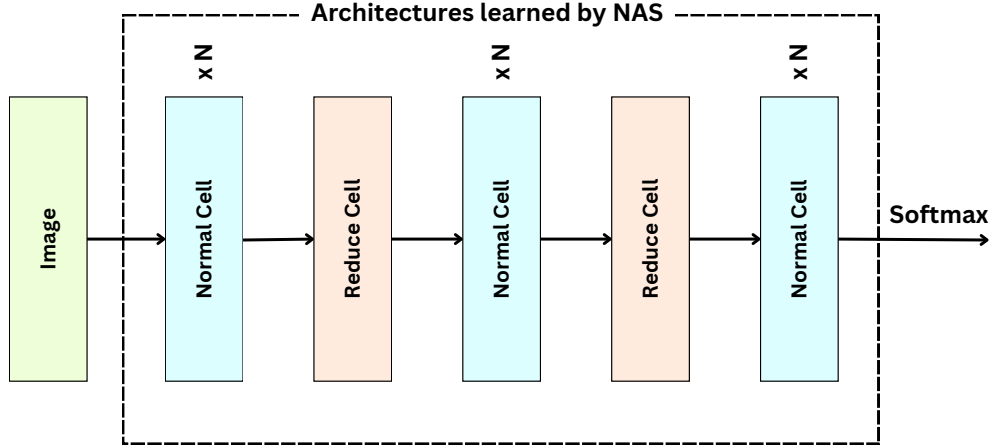


Figure 3.3: After learning the architecture of normal and reduction cells through NAS, a final network is formed by stacking multiple cells together. Cells at $1/3$ and $2/3$ of the total depth are reduction cells. The number of normal cells stacked between reduction cells, denoted as N , is a tunable parameter during the architecture search process in FAWNS.

3.3.2 Search Strategy Hub

This hub stores and manages various algorithms for exploring the model space. FAWNS directly utilizes NNI’s array of exploration strategies, enabling users to select from any of the strategies that NNI offers. NNI’s search algorithms span three primary types, each featuring unique methodologies and applications:

1. *Reinforcement learning-based NAS*: This strategy employs an RL agent to navigate the architecture space, receiving rewards based on the performance of the architectures it generates. This feedback loop guides the agent toward increasingly effective designs over time.
2. *Evolutionary algorithms-based NAS*: Utilizing genetic algorithms, this approach mimics natural selection where architectures evolve through mutations and crossovers. The fitness of each architecture, often gauged by its accuracy on a validation set, determines its survival and reproduction probabilities.
3. *Gradient-based NAS (e.g., DARTS)*: This method optimizes architecture parameters via gradient descent, allowing for a continuous relaxation of the architecture space.

Gradient-based NAS is recognized for its speed and resource efficiency when compared to the methods discussed in the bullets above.

The FAWNS framework directly incorporates NNI’s search strategies, utilizing them as-is to explore the designated search spaces. We will offer detailed explanations of the strategies we employ on the DARTS search space for our experiments in Section 4.3.2.

3.3.3 Model Evaluator

In NAS, the goal is to discover neural network architectures that perform well on unseen data. Performance estimation is critical, as it involves predicting how well these architectures will generalize. The challenge, notably however, lies in scalability: efficiently running and managing numerous trials to evaluate different architectures. In NNI, this is handled by the evaluator, which standardizes performance estimation. Such an evaluator helps in systematically managing the exploration of the search space, ensuring that multiple architecture evaluations can be run simultaneously and efficiently.

More specifically, in the exploration process of our framework, the search strategy systematically generates new models, each of which is then evaluated for performance by an evaluator handling the training and validation phases. This module supports a variety of evaluators, each tailored to different scenarios and model requirements such as classification or regression tasks. These evaluators can be further customized to fit specific search strategies and search spaces. Since WF is a classification task, we will briefly discuss NNI’s classification evaluator, which we have customized for WF.

NNI’s classification evaluator for WF: This component is designed to manage the evaluation of classification models by overseeing the training, validation, and testing processes. It sets up the optimizer and loss function, and manages the training over multiple epochs, processing data in batches to ensure efficient memory usage and faster computations. During each iteration, the loss is computed, and the optimizer adjusts the model parameters to minimize this loss. At regular intervals, the model is evaluated on the validation dataset to monitor performance, with metrics such as accuracy and loss being computed and reported back to the exploration strategy. This feedback loop guides the iterative improvement of the model. Additionally, the evaluator can implement early stopping based on validation performance to prevent overfitting and save the best-performing models at specific checkpoints.

Assembling the final architecture post-NAS: Following a systematic exploration of the defined search space using a chosen exploration strategy and evaluator, a specific con-

figuration of normal and reduction cells will be identified, as depicted in Figure 3.4. Note that while these cells were searched using NAS on an undefended Tor dataset, running FAWNS on a different dataset will likely result in a different architecture. The green blocks represent the inputs from the two preceding cells, while the blue blocks denote the four nodes of the current cell. Comparing the structure of these cells with the general DARTS cell structure shown in Figure 3.2, we observe that the NAS process has determined and set the candidate operations on the edges between the input and intermediate nodes. When an intermediate node has two input operations, their outputs are combined by element-wise addition to form a singular output. The yellow block stores the depthwise concatenation of outputs from the four intermediate nodes. To build the final architecture from the identified normal and reduce cells, the cells must be stacked in a specific sequence, typically alternating between several normal cells and a reduction cell to progressively refine and reduce feature representations as shown in Figure 3.3. The choice for the number of times the normal cells that get stacked between reduction cells, N , can vary and is a tunable parameter during architecture search in FAWNS. Choosing the value of N involves balancing model capacity and performance against computational resources and application requirements. Higher values of N can enhance performance but increase the risk of overfitting, as well as raise computational costs and inference times. Conversely, lower values of N are more efficient and less likely to overfit but might not fully exploit the neural architecture’s potential, especially for more complex datasets.

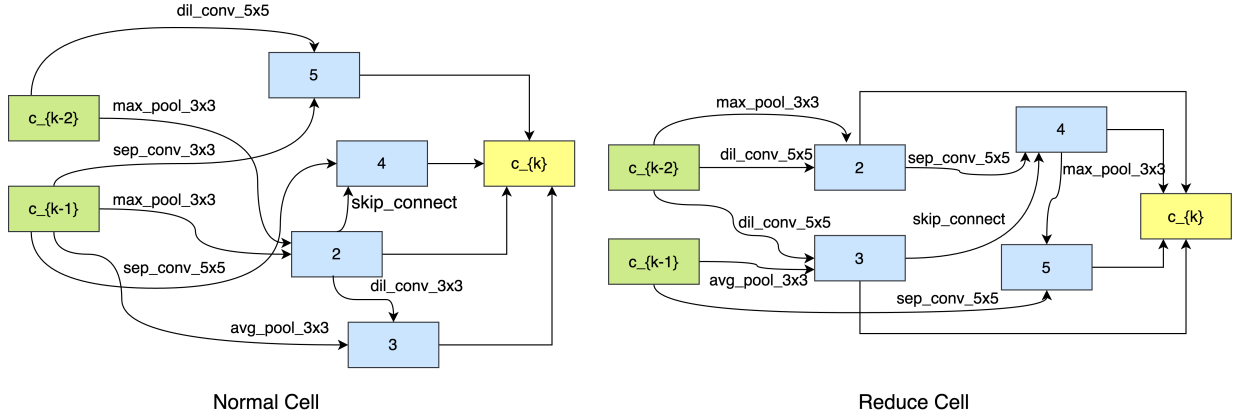


Figure 3.4: Visual representation of the optimized normal and reduce cells from the DARTS space learned using random one-shot exploration strategy.

3.3.4 NAS Experiment Engine

The **NAS Experiment Engine** serves as a central component in the NAS process within the FAWNS framework. It takes inputs such as the chosen search space, search strategy, and the model evaluator and kicks off a NAS trial. In NNI, a trial represents a single tuning process, such as one run of neural architecture search on a defined search space. This engine manages the full lifecycle of NAS trials, including overseeing trial execution, monitoring ongoing experiments, and gathering results at their conclusion. Its trial management system is designed to resume NAS trials from the last checkpoint in case of any interruptions, ensuring that no computational efforts are lost and that experiments can continue seamlessly. Upon completing model exploration, the **NAS Experiment Engine** sends the optimal final model back to the **Traffic Analyzer**'s submodule that initiated the NAS process, enabling further training and evaluation of the model architecture.

3.4 Experiment Manager

The **Experiment Manager** exposes a web portal that allows users to visualize the NAS process, helping them monitor its progress and analyze the outcomes of NAS trials. Additionally, the manager oversees multiple NAS experiments, giving users the ability to configure, start, pause, or end experiments through either a command-line interface or the aforementioned web portal as shown in Figure 3.5.

NNI also integrates with TensorBoard, enabling comprehensive visualization and monitoring of the NAS process. While NNI's web portal primarily focuses on managing NAS trials with tools to configure, start, pause, or end experiments, TensorBoard's capabilities are more attuned to the deep learning aspects of the experiment. It visualizes scalar metrics like accuracy and loss over time, presents model computational graphs for structural insights, and displays tensor value distributions through histograms to track the evolution of weights and biases during training. For instance, as seen in Figure 3.6, TensorBoard can compare multiple NAS trials, allowing for inspection of convergence trends and monitoring of hyperparameters such as learning rate adjustments.

3.5 Logger

During model training in FAWNS, we encountered obscure CUDA errors due to incorrectly shaped inputs. Debugging CUDA can be difficult because errors often stem from parallel

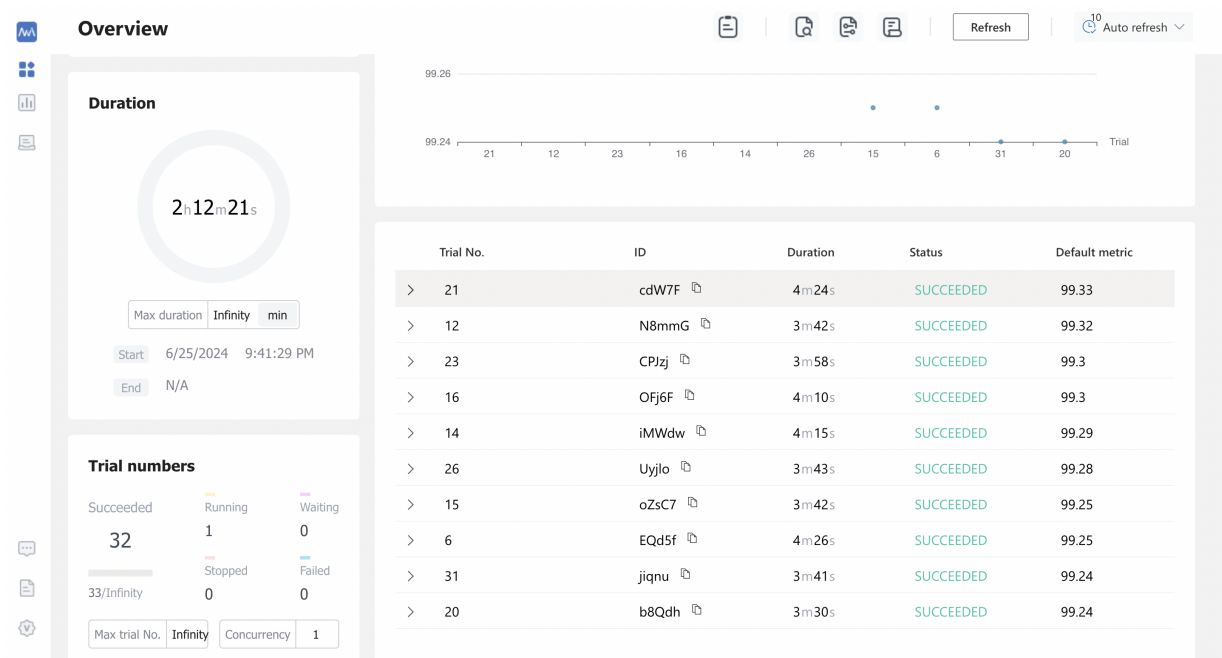


Figure 3.5: Example of NAS experiment trials on NNI’s web portal.

execution, memory management, and synchronization issues that are hard to trace. To address these issues, we devised the **Logger** component using Python’s logging module. By knowing the expected functioning of the system, including data shapes, sizes, and formats at specific code points, we log warnings or errors when abnormalities are detected.

Overall, the **Logger** plays a crucial role in the operation and incremental development of the FAWNS framework by recording all of its activities. It collects details of data preprocessing, the execution of NAS experiments, model performance checkpoints during various phases such as training and testing, and system errors—essential for troubleshooting and refining operations. The **Logger** ensures that critical information is systematically captured and can be either displayed on the console or saved to a file. **Logger**’s logs are organized into different severity levels, including informational messages, warnings, errors, and critical issues. This categorization helps users and developers easily distinguish and manage the system’s operational insights efficiently.

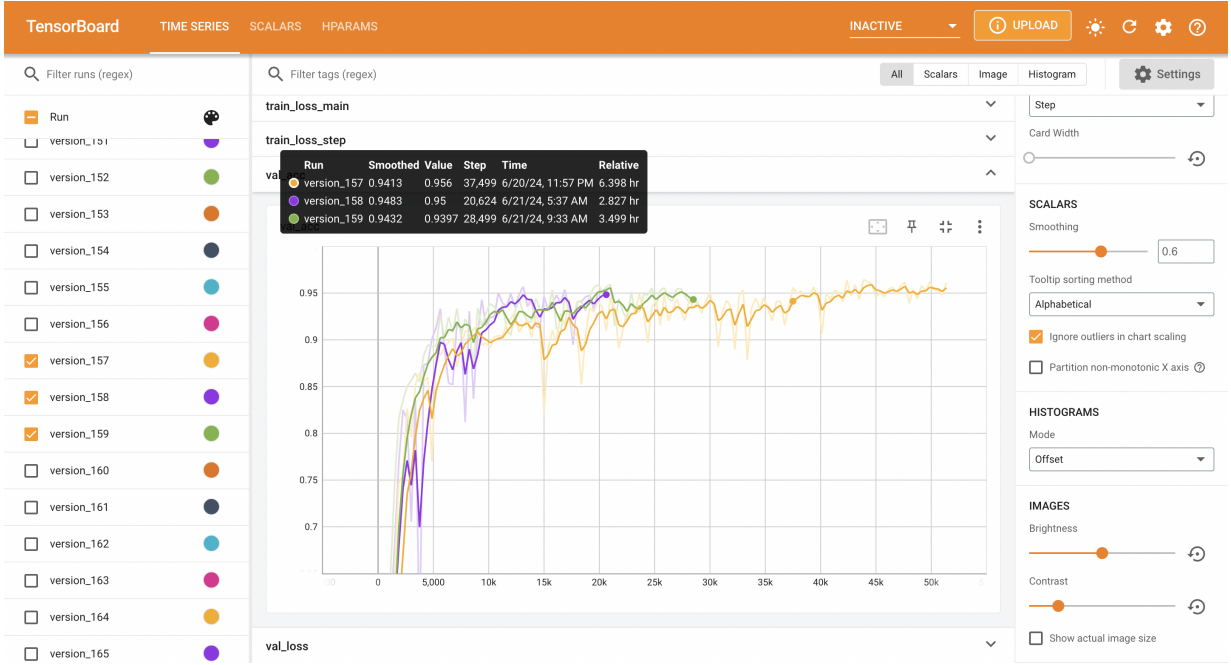


Figure 3.6: NNI NAS trials visualized on TensorBoard.

Summary

In this chapter, we introduced the architecture of the FAWNS framework. FAWNS includes key components such as the **Extractor**, which processes traffic traces and prepares datasets, and the **Traffic Analyzer**, which manages data preprocessing and splits for NAS. The **WF Attacker** submodule executes NAS, identifies optimal architectures, and handles training, validation and testing.

The framework uses the DARTS search space for effective architecture exploration with normal and reduction cells. The NNI’s **NAS Module** facilitates NAS through components like the **Model Space Hub**, **Search Strategy Hub**, and **Model Evaluator**, which work together to explore and evaluate architectures. The **NAS Experiment Engine** oversees NAS trials, ensuring efficient execution and result collection.

The **Experiment Manager** provides a web portal for monitoring NAS trials, and the **Logger** captures operational details for troubleshooting. Together, these components streamline the NAS process, enabling the development of DNNs for WF tasks.

Chapter 4

Implementation of FAWNS Modules

This chapter provides a breakdown of the various modules that realize the FAWNS framework, showcasing the technical details underlying their functionality. As such, the next few sections will cover different implementation details of these modules. Throughout this chapter, our focus is placed on the lilac-coloured modules shown in Figure 3.1, as these are either newly implemented by us, or adapted with significant modifications. Accordingly, we will not cover the green-colored modules since these were incorporated from their original sources without changes.

4.1 Extractor

The **Extractor** module in FAWNS is designed to convert raw traffic traces into simpler, more manageable trace representations, specifically focusing on direction and directional-timing representations. Our goal is to create a unified representation of traces, as datasets often store traces in different formats or use varying trace representations. By standardizing these representations, FAWNS can more effectively process and analyze the data, ensuring consistency and compatibility across different datasets. In addition, the **Extractor** also includes various functions designed to interact with WF defense simulators in order to generate simulated defended traffic based on datasets collected without any applied defense.

We implemented this module as a Python script which stores the results of the aforementioned feature extraction and manipulation operations in compressed NumPy arrays for future use. Next, we detail the implementation of the **Extractor**'s key functionalities.

Simulating defenses. We adopt publicly available defense simulators from previous research [VRB23, HH22] to simulate defenses on undefended datasets. Since some of these simulators differ in how they format their output traces, our modifications to these defense simulator codes were minimal, focusing solely on ensuring compatibility with datasets formatted according to the Wang standard, both for input and output.

Standardizing the ingestion of network traces. Different datasets may use various trace representations, necessitating a mediation layer to uniformize traces for seamless integration with FAWNS. In *website-trace* datasets like AWF and DS-19, files containing traces from monitored websites are commonly named following a $w - t$ naming convention, where w denotes the website and t denotes a given trace collected when accessing w . For instance, a file named 2 - 10 contains the 11th trace collected by visiting website 3 (taking zero-indexing into account). This naming convention allows WF researchers to quickly derive the label for each trace from its filename.

In other recent datasets like BigEnough, which include *website-webpage-trace* data, matching a trace to its label involves additional care. Specifically, the w in the Wang naming format represents the product of the number of websites and the number of webpages plus the webpage under consideration ($\text{n_website} \times \text{total_webpages} + \text{n_webpage}$). For example, a trace file named 44-2 contains the 3rd trace collected by visiting the 5th webpage among 10 webpages belonging to a website of class 5. We extract the label by calculating the modulo of w with total_webpages , thereby implementing a mapping to identify traces and their labels in *website-webpage-trace* datasets.

For traces from unmonitored websites, stored in files named t in the Wang format, all unmonitored traces are given a label that is one more than the total number of classes. Therefore, all unmonitored traces are gathered under a single class. Since the convention for representing unmonitored websites/webpages is currently the same across datasets, there is not current need for dataset-specific handling of unmonitored trace files.

Once the desired trace and label mappings are identified, we use APIs from the NumPy library to load traces that are in the format $t_i < tab > d_i$ from input trace files, where t_i represents the time and d_i denotes the direction of the i -th packet in Tor traffic (-1 for outgoing and $+1$ for incoming). These are used to form the directional-timing vector derived from the dot product of t_i and d_i .

Individual trace preprocessing. To maintain a standard feature vector length of 5000, we either truncate longer vectors or pad shorter ones with zeros. Additionally, if specified, the number of traces per website can be limited to a user-specified maximum, allowing for down-sampling or balancing of an imbalanced dataset.

With directional-timing feature vectors now ready, we discuss the role of the **Traffic Analyzer**, which enables various preprocessing operations to prepare the data for NAS.

4.2 Traffic Analyzer and WF Attacker

The **Traffic Analyzer** handles the transformation of raw traffic data into structured inputs suitable for NAS, including normalizing and segmenting the data. Its modular design accommodates one-dimensional traffic traces by reshaping them for high-dimensional NAS applications, typically designed for image processing. This allows for adaptation to new traffic analysis tasks and integration of additional submodules as needed. We now discuss the technicalities of **Traffic Analyzer**’s only current submodule, the **WF Attacker**.

As one of the central components of FAWNS, the **WF Attacker** submodule performs WF-specific feature preprocessing, prepares training, validation, and testing splits, and manages both open-world and closed-world scenarios for website fingerprinting experiments. This submodule also orchestrates the exploration of suitable deep neural network architectures using NAS, initiates the model search process, trains the optimal architecture identified by NAS, and gathers the results generated by the search, training, and testing phases. Here, we shed additional details on the implementation of its core functionalities.

Preprocessing directional-timing feature vectors. The **WF Attacker** module reshapes the one-dimensional WF directional-timing feature vectors for compatibility with existing NAS algorithms, which are typically designed for image processing tasks. Using the **NumPy** Python library, the initial one-dimensional feature vector of shape `[5000,]` is transformed by appending trailing zeros to create a 5041-element vector. This vector is then reshaped into a 71x71 matrix and expanded into three channels to mimic an RGB image, resulting in inputs of shape `[71, 71, 3]`.

Generating training-testing splits. Following the reshaping of vectors, the module generates a 9 : 1 training-testing split in a stratified manner using the `train_test_split` function from **scikit-learn**’s `model_selection` module. We use the `stratify=y` parameter to ensure that the class distribution of the labels y is preserved in both the training and testing sets. This is particularly important when dealing with imbalanced datasets, where some classes might have significantly more samples than others. By using `stratify=y`, each subset (training and testing) will have a similar proportion of samples from each class as the original dataset. The test split is reserved solely for making predictions with the final trained model.

Configuring k-fold cross-validation. To address overfitting and maximize data usage of the downsampled, smaller WF datasets, the **WF Attacker** employs k-fold stratified cross-validation to further split the training set into 9 : 1 train-validation splits. To do this, we use `StratifiedKFold` class from `scikit-learn`’s `model_selection` module.

Architecture exploration and training. During every cross-validation fold, the **WF Attacker** leverages NNI’s **NAS Module** to conduct both the architecture search and the training of the best architecture. The selected exploration strategy identifies the optimal architecture. This architecture’s weights are subsequently reinitialized and trained on the same train-validation split to develop the final model, which is then used to make predictions on the reserved test split.

The integration of NNI within FAWNS is designed to abstract NAS-specific implementation details away from the **WF Attacker** submodule. **WF Attacker** simply invokes the appropriate NAS strategy modules written using NNI APIs. For instance, to execute the DARTS strategy, the **WF Attacker** calls the `get_best_darts_model` method from FAWNS’ `search_space_hub.darts` module which initiates a NAS experiment in the DARTS search space using DARTS exploration. Given that FAWNS currently supports DARTS exploration, we detail the implementation of the `search_space_hub.darts` module in Section 4.3.1.

After the final model is built (see Section 3.3.1), it is freshly trained on the same train-validation split the model was searched on to learn the weights. The model training leverages PyTorch’s functionalities to handle batch processing, model updates, and backpropagation efficiently. The effectiveness of the final trained model is evaluated based on the accuracy of predictions in the closed-world scenario and additionally on precision, recall, and F1-score in the open-world scenario.

Enabling model checkpointing. Throughout the training process across multiple folds, **WF Attacker** checks if the current model outperforms previous ones based on test accuracy. If a new model in the next fold achieves higher test accuracy, it is saved as the best model. This is implemented with a conditional statement that compares the `best_so_far` accuracy with the current fold’s `test_results["test_acc"]` where `test_results` is a dictionary containing search, training and testing statistics from each cross-validation fold. If the current model is better, it updates `best_so_far` and saves the model using PyTorch’s `torch.save` function. The saved model includes all learned parameters and can be reloaded later for inference on a different dataset for further analysis.

Having discussed the salient functionalities of the **WF Attacker**, we will now focus on the **NAS Module** with which the **WF Attacker** collaborates to explore DNN architectures on the dataset at hand.

```

1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4
5 class SearchModel(ModelSpace): # should inherit ModelSpace rather than nn.Module
6     def __init__(self):
7         super().__init__()
8         self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)
9         self.dropout1 = nn.Dropout(0.25)
10        self.fc1 = nn.Linear(784, 10) # Assuming input size of 28x28
11
12    def forward(self, x):
13        x = F.relu(self.conv1(x))
14        x = F.max_pool2d(x, 2)
15        x = torch.flatten(x, 1)
16        x = self.dropout1(x)
17        output = self.fc1(x)
18        return F.log_softmax(output, dim=1)
19
20 # Initialize the base model
21 base_model = SearchSpace()

```

Listing 1: Initializing a base model in PyTorch.

4.3 NAS Module

NNI’s NAS Module, integrated with FAWNS, is structured around four key submodules discussed in Section 3.3: the model space hub, which defines various model configurations; the Search Strategy Hub, which determines how the search space is explored; the model evaluator, which assesses the performance of generated models; and the NAS Experiment Engine, which orchestrates the entire NAS process. Implementation specifics for each of these submodules are detailed below.

4.3.1 Search Space

The search space contains a predefined set of building blocks and operations that can be used to construct neural network architectures. In FAWNS, the integration with NNI allows us to define a model space that consists of two parts: a base model and potential mutations of that base model. Creating a base model in FAWNS mirrors the process of defining a model in frameworks like PyTorch or TensorFlow. Listing 1 shows a straightforward example of how to define a base model.

Although a base model is a concrete instantiation of a PyTorch (or TensorFlow) model, it alone does not constitute a model search space. NNI equips users with primitives to

```

1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4 from nni.nas.pytorch import ModelSpace, MutableDropout, LayerChoice
5
6 class SearchSpace(ModelSpace): # Should inherit ModelSpace rather than nn.Module
7     def __init__(self):
8         super().__init__()
9         # conv1 gets to choose between two layers
10        self.conv1 = LayerChoice([
11            nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1),
12            nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=2)
13        ], label='conv1_kernel')
14        # Dropout layer gets to choose one of the dropout values
15        self.dropout1 = MutableDropout(nni.choice([0.0, 0.25, 0.5]))
16        self.fc1 = nn.Linear(784, 10)
17
18    def forward(self, x):
19        x = F.relu(self.conv1(x))
20        x = F.max_pool2d(x, 2)
21        x = torch.flatten(x, 1)
22        x = self.dropout1(x)
23        output = self.fc1(x)
24        return F.log_softmax(output, dim=1)
25
26 # Initialize the model space
27 model_space = SearchSpace()

```

Listing 2: Initializing a model space in PyTorch using NNI’s mutation APIs.

express mutations on the base model, thus enabling the creation of a model space that includes numerous models. Based on the above base model, and armed with NNI’s APIs, we can define a model space as described in Listing 2.

Here, the model space definition employs two mutation APIs, `nn.LayerChoice` and `nni.choice()`. The `nn.LayerChoice` function accepts a list of candidate modules and integrates one chosen module into each sampled model, functioning similarly to a standard PyTorch module. `MutableDropout` allows the Dropout layer to be mutable, meaning its parameters (such as dropout rate) can be adjusted and optimized during the NAS process. The `nni.choice()` function sets the dropout rate as a parameter of `MutableDropout`.

Since this thesis focuses on exploring the DARTS search space, we provide an example of how to construct a separable convolution with a 5x5 filter, which is one of the operations supported in the DARTS search space detailed in Section 3.3.1. This can be implemented using NNI’s mutation APIs as shown in Listing 3.

The code in Listing 3, adopted from the source package of Liu et al.’s unsupervised neural architecture search (UnNAS) project [LDH⁺20] and NNI’s `nasnet` module, defines

```

1 class SepConv(nn.Sequential):
2
3     def __init__(self, C_in, C_out, kernel_size, stride, padding, affine=True):
4         super().__init__(
5             nn.ReLU(inplace=False),
6             MutableConv2d(
7                 C_in, C_in, kernel_size=kernel_size, stride=stride,
8                 padding=padding, groups=C_in, bias=False
9             ),
10            MutableConv2d(C_in, C_in, kernel_size=1, padding=0, bias=False),
11            MutableBatchNorm2d(C_in, affine=affine),
12            nn.ReLU(inplace=False),
13            MutableConv2d(
14                C_in, C_in, kernel_size=kernel_size, stride=1,
15                padding=padding, groups=C_in, bias=False
16            ),
17            MutableConv2d(C_in, C_out, kernel_size=1, padding=0, bias=False),
18            MutableBatchNorm2d(C_out, affine=affine),
19        )
20
21     darts_operations = {
22         'sep_conv_5x5': lambda C, stride, affine:
23             SepConv(C, C, 5, stride, 2, affine=affine)
24     }

```

Listing 3: Creating a mutable separable convolutional operation with NNI’s mutation APIs.

a custom sequential neural network module called **SepConv**, which stands for “Separable Convolution”. It also defines a dictionary **darts_operations** that maps operation names to lambda functions that create instances of the **SepConv** class with specific parameters. Other operations supported by the DARTS space, specified in Section 3.3.1, can be defined similarly and added to **darts_operations** to create a dictionary of all candidate operations of the DARTS search space.

In the process of adapting search spaces originally designed for image datasets like CIFAR and ImageNet to our WF datasets, several key modifications were necessary. For instance, CIFAR and ImageNet models often utilize filter sizes that are optimized for their typical image dimensions, which are around 32x32 pixels. Given that the preprocessed website fingerprinting data is transformed into 71x71 dimensions, adjustments to the convolutional layer filter sizes and configurations are needed. This adaptation ensures that the filters are appropriately sized to capture the larger spatial dimensions and potentially more complex patterns in the reshaped traffic data. Specifically, filter sizes, strides, or padding may need to be altered to optimize feature extraction for the larger input size.

4.3.2 Search Strategies

NNI provides a broad range of built-in algorithms spanning three primary types of strategies outlined in Section 3.3.1 to facilitate the search for optimal neural network architectures. The FAWNS framework directly incorporates NNI’s search strategies, utilizing them as-is to explore our designated search spaces. Along with our search strategies, we employ gradient clipping to mitigate the problem of exploding gradients. This is achieved by setting a maximum threshold for the magnitude of the gradients; any gradients exceeding this threshold are clipped to the specified value, thereby stabilizing the training process and preventing large updates that could cause the model to diverge.

Choice of search strategies. We use the following search strategies to explore the DARTS search space for the experiments discussed in this thesis:

- *Random one-shot* search strategy, which trains the supernet with uniform path sampling, meaning during each training iteration, one path is chosen from the supernet with equal probability. Thus, every single path has an equal chance of being selected for training in that particular iteration.
- *DARTS* search strategy, which involves a differentiable approach to NAS. The search space is represented as a continuous relaxation of the architecture, allowing the use of gradient-based optimization methods to search for the best architecture. The search process adjusts the weights of the softmax on each node, and the operation with the highest probability is chosen for the final structure.
- *Gumbel-DARTS*, which enhances the DARTS technique by incorporating Gumbel-Softmax, allowing for more efficient sampling of discrete architectures from the continuous space. This technique improves the search efficiency and helps in finding better-performing architectures.

Iterative model refinement. In the exploration process of our framework, the strategy systematically generates new models, each of which is then evaluated for performance by a model evaluator handling the training and validation phases. The results of these evaluations—specifically the performance metrics such as validation accuracy and loss—are fed back to the exploration strategy, enabling it to refine and generate improved models iteratively. Next, we discuss the implementation of the evaluation strategy used in FAWNS.

4.3.3 Evaluation Strategy

As discussed in Section 3.3, an evaluator ameliorates the challenges associated with efficiently training and evaluating the performance of many architectures from the search space. In other words, an evaluator standardizes performance estimation. Evaluators can range from simple configurations, like standard training and validation, to more complex setups, allowing for flexibility and scalability in assessing model performance across multiple NAS trials. This helps in systematically managing the exploration of the search space, ensuring that multiple architecture evaluations can be run simultaneously and efficiently.

Customizing the NNI Evaluator for WF. For users’ convenience, NNI provides some commonly used model evaluators built upon the PyTorch-Lightning library such as the `ClassificationModule` evaluator for classification tasks and `RegressionModule` for regression tasks. For our classification task within the NNI NAS framework, we customize our model evaluator based on the framework’s built-in `ClassificationModule`. To maintain congruency with NNI’s implementation of the `ClassificationModule`, this evaluator is also implemented in the PyTorch Lightning¹ style. For WF, this customization involves using the `nn.CrossEntropyLoss` class as the loss criterion, which is suitable for multi-class classification tasks, tailored to the configurable number of classes based on the dataset. The evaluator adjusts the number of epochs for both searching and training, allowing for optimal tuning on the chosen dataset. It uses stochastic gradient descent (SGD) optimizer with momentum, and a cosine annealing scheduler to adjust the learning rate based on the epoch number for faster convergence and avoiding local minima.

Optimizing model performance with callback metrics. To effectively monitor and optimize model performance during training and validation phases, our customized evaluator utilizes a set of callback metrics provided by NNI’s `ClassificationModule`. These metrics include `train_loss`, `train_acc`, `val_loss`, and `val_acc`. By monitoring these metrics, one can identify if and when the model begins to overfit, underfit, or if it has reached an optimal state. Logging `val_loss` also helps us implement early stopping to prevent overfitting by terminating training once the validation loss ceases to decrease, despite improvements in training loss.

In summary, the model evaluator is essential in the FAWNS framework, as it streamlines the training and evaluation of multiple neural architectures. This is achieved by leveraging NNI’s customized `ClassificationModule` tailored for WF. Having discussed

¹PyTorch Lightning is a lightweight PyTorch wrapper that organizes PyTorch code to decouple the research code (model definition, training, validation) from engineering code (checkpointing, logging, training loops) and therefore, this coding style is regarded as cleaner and more maintainable.

```

1 from nni.nas.experiment import NasExperiment
2 experiment = NasExperiment(model_space, evaluator, strategy)
3 experiment.run()
4 abstract_arch = experiment.export_top_models(formatter='dict')[0]
5
6 from nni.nas.space import model_context
7 with model_context(abstract_arch):
8     final_model = DartsSpace(
9         width=36, # the number of nodes/channels in each cell layer
10        num_cells=20, # total number of cells in the network
11        dataset="awf", # dataset for which the architecture is optimized
12        targets=targets, # target classes or output dimensions
13        auxiliary_loss=True, # auxiliary heads for training stability?
14        drop_path_prob=0.2) # dropout probability to regularize the network
15 return final_model

```

Listing 4: Scaling up the architecture selected by the search process. Note that the dataset parameter is set to “awf” to indicate the particular WF dataset being used.

the three main dimensions of NAS and their implementation in FAWNS, we now detail how these components can be integrated with the **NAS Experiment Engine** to initiate the NAS process.

4.3.4 NAS Experiment Engine

The code in Listing 4 effectively orchestrates the NAS process using the **NAS Experiment Engine**. It initializes the **NasExperiment** with the defined **model_space**, **evaluator**, and **strategy**, and then runs the experiment to search for the best-performing architecture. Once the experiment completes, the best architecture is exported and instantiated using the **model_context** function, translating the abstract architecture into a concrete model, parameterized by specific attributes like width and the number of cells (**DartsSpace**). This model is then trained and evaluated on the specified dataset.

While training, regularization techniques such as auxiliary loss and drop path probability are incorporated. The auxiliary loss, calculated using intermediate outputs from the model and combined with the primary loss, helps stabilize training and improve convergence. Drop path probability, where entire layers or paths within the neural network are randomly dropped during training, forces the network to learn more robust features by preventing co-adaptation of layers.

4.4 Logger

FAWNS utilizes Python’s `logging` module to manage updates, warnings, and other critical diagnostics. This integration ensures that important information is systematically captured and can be output to the console or written to a file. Logs are categorized into different levels as provided by the Python module, such as informational messages, warnings, errors, and critical issues. This structure allows users to easily differentiate and manage the system’s operational insights effectively.

The `Logger` class serves as the primary interface for logging. While it is possible to create multiple loggers within an application, we use a single instance of the `logger` class throughout the FAWNS framework. This instance is configured with a `StreamHandler` and a `FileHandler`, allowing it to send log messages to both the console and a file.

In Python’s `logging` module, log levels are used to indicate the severity of a log message. Each log message is associated with a level that represents its importance. The logging module defines the following standard log levels, in increasing order of severity and we adopt the same in FAWNS:

1. **DEBUG**: Provides detailed diagnostic information useful for debugging issues, such as mismatched input data dimensions.
2. **INFO**: Confirms that operations are working as expected, such as successful dataset loading, NAS experiment initialization, and data export.
3. **WARNING**: Indicates unexpected events or potential problems while the software continues to function. For instance, FAWNS may log a warning when the number of training epochs is unusually low, suggesting the model might be undertrained.
4. **ERROR**: Signals serious issues that prevent normal operation, such as missing data files or failed model checkpoints.
5. **CRITICAL**: Highlights very serious errors that may halt the application, such as CUDA memory failures during model training, preventing further experiment execution.

Additionally, FAWNS tracks and outputs fold-level training and testing statistics, which include training/validation accuracy and loss during both the architecture searching and model training phases. This detailed performance data is written to disk using Python’s `pickle` module, facilitating in-depth analysis and review of the model’s behavior across different stages of the learning process. At the end of all folds, a summary of the model’s performance across all folds is logged, providing insights such as the average and best accuracy, total number of parameters, and detailed test accuracy values for each fold.

Chapter 5

Evaluation

In this chapter, we aim to showcase the benefits of using FAWNS to obtain deep learning models which show high accuracy in website fingerprinting tasks. We begin by outlining our evaluation methodology in Section 5.1 and our laboratory testbed in Section 5.2. In Section 5.3, we discuss how we optimized the hyperparameters for the DARTS-inspired search strategies used throughout our experiments. Section 5.4 describes the results of an initial round of experiments where we benchmark the accuracy of models generated by FAWNS when targeting undefended Tor traces. Next, in Section 5.5, we analyze FAWNS’ accuracy on defended Tor traffic. Finally, in Section 5.6, we evaluate and discuss the generalizability of FAWNS-generated architectures on datasets that were not used as part of the original NAS-based model search procedure.

5.1 Evaluation Methodology

This section starts by enumerating the goals of our evaluation. Then, it provides details about the WF scenario we consider, our model training procedure, and the datasets used throughout our experiments.

Evaluation goals. The goals of our evaluation are three-fold:

1. Assess the accuracy obtained by NAS-generated architectures on *undefended* Tor traces, in the closed-world scenario.
2. Assess the accuracy obtained by NAS-generated architectures on *defended* Tor traces, in the closed-world scenario.

3. Assess the applicability of NAS-generated architectures to other WF datasets which were not used in the original NAS process.

To achieve the above goals, we first assess the ability of FAWNS to generate architectures that can perform well when launching WF attacks over a prominent undefended dataset (described further in this section). We compare the obtained results to those of a popular WF attack in order to establish a proper baseline. Second, we evaluate the performance of the same models against defended traces, based on the same undefended traces, by applying simulators of exiting WF defenses (as mentioned before in Section 4.1). Lastly, to evaluate the generalizability of the NAS-generated architectures, we devise experiments that aim to exercise the models’ prediction behavior on larger datasets following a similar distribution, as well as on datasets that capture concept drift over multiple weeks.

Next, we provide additional details on the datasets we used and on how we configured our model’s training and testing procedures.

Evaluation scenario and datasets. In our evaluation, we assume the closed-world WF scenario. To conduct our experiments, we leverage different variations of the closed-world AWF dataset [RPJ⁺18]. The most basic dataset we use is AWF100, which contains 100 traces for each of 100 different websites visited via Tor. These website traces contain directional and timing data related to Tor cells (estimated from Tor packets [WG13]). Each trace is either truncated to 5000 cells or padded with zeros if its original length is shorter. We leverage the same dataset for generating defended Tor traces using a set of defense simulators [GZZW23, HH22]. We also use alternative versions of the AWF dataset when assessing the generalizability of architectures generated by FAWNS. In particular, we leverage the AWF200 and AWF450 datasets (which contain larger numbers of traces per class) to assess the performance of models trained on a sub-sampled dataset. We also use alternative versions of AWF100 collected over a period of six weeks to assess the models’ resilience against concept drift.

Model selection, training, and evaluation procedure. We split the data into 90% training and 10% testing sets, ensuring that every model is tested on the same data split. The training set is further divided into 90% training and 10% validation in each cross-validation fold used throughout our experiments. This data is used for two fundamental processes. First, the above splits are used to search model architectures via NAS. Once the choice of the composition of normal and reduction cells is finalized, the final neural network model is formed by repeatedly stacking these cells. Second, the splits are used to train the selected model and learn adequate weights. The trained model is then used to make predictions on the held-out test set, with the accuracy of these predictions serving as our main performance metric.

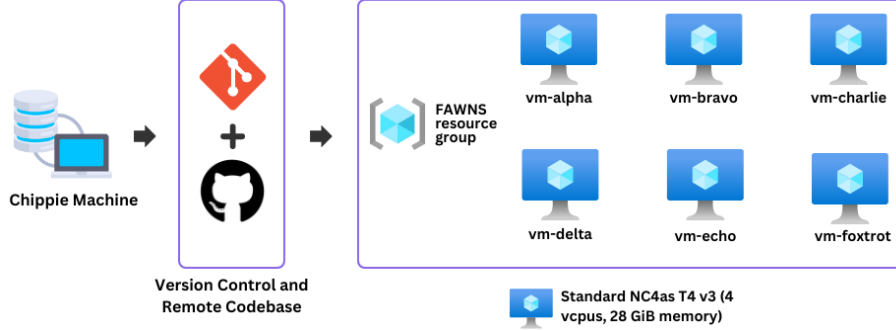


Figure 5.1: An overview of FAWNS’ testing infrastructure.

Generation of defended traffic. Inspired by Veicht et al. [VRB23], we assess the effectiveness of Tik-Tok and FAWNS-generated models on a set of relevant WF defenses, each accompanied by high-fidelity simulators [GZZW23]. We use these simulators to generate defended traffic from the pre-recorded undefended traces contained in the AWF100 dataset. Specifically, we employ the WTF-PAD, FRONT, Tamaraw, and RegulaTor defenses.

Defense configurations. We follow the methodology of Veicht et al. [VRB23] and use default parameters for WF defenses, as suggested in their original papers. We used the WTF-PAD implementation from the WFES repository [Che17], and two versions of FRONT [GW20] with parameters: $N_c = N_s = 1700$ and $W_{min} = 1$, $W_{max} = 14$ for FRONT_T1, and $N_c = N_s = 2500$ for FRONT_T2. Due to its larger sampling window, FRONT_T2 induces more dummy packets in the trace when compared to FRONT_T1. For Tamaraw [CNW⁺14], we used $\rho_{out} = 0.04$ and $\rho_{in} = 0.012$ with $L = 50$. For RegulaTor [HH22], we used a padding budget of 3550, initial rate of 277, decay rate of 0.94, burst threshold of 3.55, upload packet ratio of 3.95 and delay cap of 1.77.

Comparison baseline. In our evaluation, we compare the effectiveness of the NAS-generated models with the Tik-Tok [RSM⁺20] attack. We chose Tik-Tok as it is a typical choice for benchmarking new attacks that recent WF defense proposals aim to protect against [MHO⁺23], and because of its reliance on full trace information (both directional and timing-related data). We use the default Tik-Tok architecture (i.e., the DF model with directional-timing vectors as input) and parameters in our experiments, as these have previously shown good results when targeting the AWF dataset.

5.2 Laboratory Testbed

This section showcases the composition of the laboratory testbed where we performed our experiments (see Figure 5.1). We start by describing our *development* infrastructure where most of FAWNS codebase was prototyped, before describing our *production* infrastructure, where FAWNS and Tik-Tok were deployed and executed towards obtaining the results that allow us to perform comparative benchmarks throughout our evaluation. Then, we describe how we synchronized our code development between the two infrastructure environments.

Development infrastructure. A Linux Container (LXC) deployed on a machine within the University of Waterloo’s Chippie computing cluster serves as the centralized node for all our development activities. It is equipped with 2 AMD EPYC 7272 12-Core CPUs and an NVIDIA A40 GPU, providing the necessary computational power to develop and test the FAWNS codebase. We test changes to the FAWNS codebase on the Chippie machine to ensure they are free of errors before deployment.

Production infrastructure. Our production infrastructure comprises six VMs deployed on Azure, each configured with a Standard NC4as T4 v3 (4 vCPUs, 28 GiB memory) and additional 128GB disk drives. These VMs are named `vm-alpha`, `vm-bravo`, `vm-charlie`, `vm-delta`, `vm-echo`, and `vm-foxtrot`, and are part of a FAWNS-specific Azure resource group. These machines play a crucial role in running experiments with the FAWNS codebase, as we leverage their accelerated GPU capabilities for enhanced data processing tasks.

Centralized code development and synchronization process. In order to maintain a stable and up-to-date codebase across our development and deployment environments, we have established a centralized code development and synchronization process. This approach ensures that all code changes are tested before deployment and that updates are seamlessly propagated across our infrastructure. The process involves the following steps:

Development on Chippie. All code changes are developed and tested on the Chippie machine to ensure stability before being considered for deployment. For version control, we utilize Git, with GitHub serving as the remote storage repository. After developing and testing code changes on the Chippie machine, we push these changes to our GitHub repository. This repository facilitates efficient collaboration and tracking of modifications.

Synchronizing Azure VMs: Each Azure VM performs a Git pull operation to fetch the latest code changes from the GitHub repository, ensuring that all VMs remain synchronized with the most recent version of the codebase. To streamline the synchronization of code changes across the Azure VMs, one can also employ continuous integration and deployment strategies using GitHub Actions.

Table 5.1: Hyperparameter space and final selections for DARTS search space optimization.

Hyperparameters for DARTS search space	Choices	Final Choice
Search Space Width	[16, 26, 36, 46]	16
Search Space Cells	[8, 14, 20, 26]	8
Train Space Width	[16, 26, 36, 46]	36
Train Space Cells	[8, 14, 20, 26]	20
Search Epochs	[10, 30, 50]	30
Train Epochs	[100, 300, 600]	600

Having outlined our evaluation methodology and laboratory testbed, we now proceed to detail the experiments conducted to assess the effectiveness of FAWNS-generated models.

5.3 Optimizing Hyperparameters for NAS Searches

Hyperparameter tuning is a fundamental process in supervised classification, involving the adjustment of model parameters, such as the number of hidden layers in a CNN. Fortunately, NAS already intrinsically tunes the internal hyperparameters of DNN models. However, when conducting a NAS search, one also has to configure a set of hyperparameters responsible for bootstrapping and guiding the NAS optimization process itself. Given the numerous hyperparameters in the NAS experiment like the learning rate scheduler, initial learning rate, dropout probability etc, an exhaustive search becomes computationally prohibitive. Therefore, our goal is to find a sufficiently good model by tuning a few key hyperparameters (listed in Table 5.1), acknowledging that these could be optimized further (e.g., through a grid search, or using classical hyperparameter tuning solutions, applied to NAS-search parameters). Due to resource and time constraints, our hyperparameter choices are based on a combination of good generalizability on unseen data, reasonable computational utilization, and faster training times.

Cascaded hyperparameter search process. To select the hyperparameters for our NAS search process, we perform a cascaded search through the hyperparameter space depicted in Table 5.1. This method involves a stepwise search across a pre-defined hyperparameter grid, prioritizing one set of parameters at a time while holding others constant. The levels of this cascade are detailed below.

Initial search over architectural dimensions. We commenced by exploring combinations of the search space width and search space cells, which define the structural complexity of our model. During this phase, other hyperparameters, including the widths and cells of the training space and the number of epochs for both searching and training, were held constant. We decided to set the train space width and train space cells to the maximum values in the range, specifically 46 and 26, respectively. Intuitively, training deeper (26 cells) and wider (46 filters) networks for a longer duration should yield better results [LSY19]. Consequently, we opted for 50 search epochs and 600 train epochs. Through systematic experimentation, we determined that a search space width of 16 and search space cells of 8 produced DNNs that performed best in terms of validation loss.

Refinement of training architecture. Subsequent to establishing the optimal search architecture, we fixed the search space width and cells to 16 and 8, respectively, and shifted our focus to optimizing the training space width and cells. This phase of the cascade aimed to balance the model’s capacity and its ability to generalize. After a series of evaluations, we identified that a training space width of 36 and training space cells of 20 provided the best accuracy on unseen validation data while maintaining a manageable computational load. Training space width and cells greater than these values resulted in models that required significantly heavier computational loads.

Final tuning of epochs. With the architectural dimensions established, the final phase involved tuning the search epochs and train epochs. Here, we aimed to optimize the duration of both the architecture search phase and the subsequent full model training. By adjusting these epochs, we found out that 30 search epochs and 600 train epochs offered a reasonable balance between thorough exploration of the architectural space and learning of the underlying data patterns, and hence, providing a good trade-off between the model’s accuracy and efficiency.

We carried out this hyperparameter search process on undefended Tor traces from AWF100. By the transferability property of neural networks [YCBL14], the hyperparameters identified from this process (search space width of 16, 8 search space cells, train space width of 36, 20 train space cells, 30 search epochs and 600 train epochs) were then used to train models on defended datasets as well. Next, we present our main findings when using these parameters to guide the search of DNNs on AWF100 undefended website traces (Section 5.4), before describing our results when generating DNNs which target defended Tor traces (Section 5.5).

Table 5.2: Comparison of search strategies, detailing their respective search and train space sizes, epochs, and validation accuracy reported with Bernoulli’s error estimate at a 95% confidence interval.

Search Strategy	Search Space	Train Space	Search Epochs	Train Epochs	Validation Accuracy (%)
Tik-Tok	-	-	-	30	97 ± 1
Random One-Shot	(16, 8)	(36, 20)	30	600	97.7 ± 0.9
DARTS	(16, 8)	(36, 20)	30	600	94 ± 1
Gumbel-DARTS	(16, 8)	(36, 20)	30	600	97 ± 1

5.4 Benchmarking FAWNS on Undefended Traces

In this section, we evaluate the performance of NAS-generated DNNs in the closed-world scenario using the undefended AWF100 dataset. Furthermore, we compare the accuracy of these models with that of the Tik-Tok model on the same undefended AWF100 dataset. Since the DF model (the DNN architecture used by Tik-Tok) converges around 30 epochs [SIJW18, SANM⁺23], we also train the model for this period when building our benchmark. Based on our hyperparameter search, we found that NAS models searched on the DARTS search space converge around 600 epochs, and adopted the same number in our evaluation.

5.4.1 Accuracy of FAWNS-generated Models

Table 5.2 presents a comparison of various search strategies, their respective search and training spaces, the number of search and training epochs, and their validation accuracy reported with Bernoulli’s error estimate at a 95% confidence interval.¹ The Tik-Tok model, which serves as a benchmark, achieves a validation accuracy of 97% with 1% error upon converging after 30 epochs of training. Below, we present our main findings.

Random-one shot strategy attains similar accuracy to Tik-Tok. The Random One-Shot (**roneshot**) strategy, utilizing a search space of (16, 8) and a training space of

¹The Bernoulli’s error estimate is calculated using the standard error of a proportion, which is derived from the formula $\sqrt{\frac{p(1-p)}{n}}$, where p is the accuracy and n is the sample size. The 95% confidence interval indicates that we can be 95% confident that the true accuracy lies within this range. This error estimate serves to quantify the uncertainty in the reported accuracy, providing a measure of the reliability and variability of the model’s performance.

Table 5.3: Comparison of search and train times per epoch for different search strategies.

Search Strategies	Search Time/Epoch (s)	Train Time/Epoch (s)
Random One-Shot	45-55	110-120
DARTS	520-530	200-210
Gumbel-DARTS	520-530	150-160

(36, 20), achieves an accuracy of $97.7\% \pm 0.9\%$ after 600 training epochs, while Tik-Tok achieves $97\% \pm 1\%$. The difference between these accuracies is not statistically significant.

DARTS requires better hyperparameter tuning. Despite using the same search and training spaces, as well as the same number of epochs as the Random One-Shot strategy, the DARTS strategy achieves a lower validation accuracy of $94\% \pm 1\%$. This suggests that DARTS may require more careful tuning of hyperparameters such as the learning rate, weight decay, and gradient clipping value to improve accuracy. In contrast, Tik-Tok and Random One-Shot achieve higher and comparable accuracies of $97\% \pm 1\%$ and $97.7\% \pm 0.9\%$, respectively. Further hyperparameter tuning for strategies like DARTS is discussed as a potential future work direction in Section 6.

Gumbel-DARTS outperforms DARTS. The Gumbel-DARTS strategy shows strong performance with an accuracy of $97\% \pm 1\%$, outperforming DARTS, which achieves $94\% \pm 1\%$. Additionally, Gumbel-DARTS performs as well as Tik-Tok without a significant difference, with Tik-Tok achieving $97\% \pm 1\%$. The Gumbel-DARTS method incorporates the Gumbel-Softmax trick to improve the sampling of discrete architectures, likely contributing to its higher accuracy compared to DARTS.

The comparison of NAS search strategies on undefended AWF traces reveals that Tik-Tok and Random One-Shot strategies achieve high and comparable accuracies. While DARTS requires better hyperparameter tuning to reach its full potential, Gumbel-DARTS demonstrates performance improvements, likely due to the use of Gumbel-Softmax.

5.4.2 Assessing FAWNS-generated Models’ Training Costs

Although one-shot search techniques like DARTS and random one-shot methods, which relax the search space to be continuous, are significantly faster than reinforcement learning (RL) or evolutionary multi-trial techniques that treat NAS as a black-box optimization problem over a discrete domain (requiring a large number of architecture evaluations),

NAS-generated models are often very large. These models can have over 20 million parameters, leading to much longer training times compared to smaller models like DF architecture [SIJW18] (that Tik-Tok adopts), which has about 3 million parameters. Calculating gradients for a larger number of parameters is computationally intensive, making each back-propagation step slower. Each training step requires more operations as more parameters need to be updated, which takes longer to process. Moreover, the choice of search strategy also significantly impacts the training time. More sophisticated strategies like DARTS and Gumbel-DARTS generally take longer to explore the search space than simpler approaches such as the random one-shot searching technique as shown in Table 5.3.

The random one-shot strategy has a significantly lower search time per epoch (45-55 seconds) compared to DARTS and Gumbel-DARTS, both of which range between 520-530 seconds. In terms of training times per epoch, random one-shot is faster (110-120 seconds) compared to DARTS (200-210 seconds). Gumbel-DARTS, while quicker than DARTS (150-160 seconds), still takes more time than random one-shot. The significant difference in search times highlights a trade-off between efficiency and thoroughness. DARTS and Gumbel-DARTS, which convert the search space into a continuous domain and use gradient-based optimization, are more thorough but come at the cost of increased search times. In contrast, random one-shot, with its lower search and training times, allows for faster iterations and quicker discovery of good architectures.

5.4.3 Integrating Early-stopping on FAWNS-generated Models

Considering the time-intensive nature of training NAS models, for our more extensive NAS experiment tuning and convergence analysis (Section 5.3) and experiments to determine generalizability (Section 5.6), we opted to use the random one-shot exploration strategy. To compare the efficacy of different search strategies, we incorporated early stopping in our model training algorithm with a patience² of 10, halting the training process if the validation accuracy did not improve over 10 training epochs. This approach effectively reduced the exploration and training time for these search strategies. We will apply this early stopping logic to our study on undefended Tor traces next.

Impact of early stopping on NAS strategies. The results in Table 5.4 provide a comparison of validation accuracy of different search strategies on undefended AWF traces, both with and without early stopping. Tik-Tok, evaluated only without early stopping, achieves an accuracy of $97\% \pm 1\%$, serving as a benchmark. The random one-shot strategy

²Early stopping patience refers to the number of epochs the training process will continue without seeing improvement in a monitored metric (e.g., validation accuracy or loss) before stopping early.

Table 5.4: Validation accuracy (maximum across 3 cross validation folds) reported with Bernoulli’s error estimate at 95% confidence interval with and without early stopping on undefended AWF traces.

Search Strategy	Accuracy (%)	Accuracy (%)
	with early stopping	without early stopping
Tik-Tok	-	97 ± 1
Random One-Shot	72 ± 3	97.7 ± 0.9
DARTS	87 ± 2	94 ± 1
Gumbel-DARTS	88 ± 2	97 ± 1

shows significantly better accuracy without early stopping ($97.7\% \pm 0.9\%$) compared to with early stopping ($72\% \pm 3\%$), indicating the necessity of extended training. DARTS performs better without early stopping ($94\% \pm 1\%$) than with it ($87\% \pm 2\%$), and Gumbel-DARTS also improves without early stopping ($97\% \pm 1\%$ vs. $88\% \pm 1\%$). These results indicate that all three strategies—random one-shot, DARTS, and Gumbel-DARTS—necessitate longer training periods for optimal performance. Additionally, the results might suggest that a patience value of 10 is too low for the model’s performance to recover during training after a small dip in validation accuracy. If early stopping is to be used, the patience parameter should also be tuned in accordance with the dataset and defense to avoid prematurely halting training.

5.5 Attacking Defended Traces with FAWNS

Here, we evaluate WF attacks against defenses after training models for a full 600 epochs without early stopping, and then, with early stopping. This dual approach will allow us to comprehensively assess how different training durations impact the effectiveness of these models against various defenses.

5.5.1 Attacking Defended Traces without Early-stopping

From Table 5.4, it was clear that the accuracy values with early stopping on undefended datasets, were much lower than the accuracy values attained with the complete 600 epochs of training. This prompted us to conduct NAS experiments with 600 epochs of training

Table 5.5: Validation accuracy from NAS models trained with the random one-shot strategy for 600 epochs on DARTS search space, compared with Tik-Tok trained for 600 epochs. This table includes the Bernoulli error estimates at a 95% confidence interval calculated on the best validation accuracy across 3 cross-validation folds.

Defense	DARTS_roneshot (%)	Tik-Tok (%)
NoDef	97.7 ± 0.9	96 ± 1
Front_T1	70 ± 3	51 ± 3
Front_T2	59 ± 3	34 ± 3
Tamaraw	3 ± 1	3 ± 1
WTF_PAD	81 ± 3	75 ± 3
RegulaTor	19 ± 3	13 ± 2

using the random one-shot strategy to get a comprehensive understanding of FAWNS-generated models’ expected performance on defended traces. The early stopping logic, although helpful in saving computational resources and training time, was limiting. As detailed in the section on training costs for different one-shot strategies, we selected the random one-shot approach due to its quicker search and training times. While Tik-Tok typically converges around 30 epochs and would not benefit from additional training, we extended its training to 600 epochs for a fair comparison. The results of these experiments are presented in Table 5.5, with the convergence of both models illustrated in Figure 5.2. It should be noted that, in contrast to Tik-Tok, whose architecture remains the same for all defenses, FAWNS generates a distinct architecture for each defense.

DARTS_roneshot markedly improves over Tik-Tok with 600-epoch training.

The DARTS_roneshot model demonstrates significant performance improvements across nearly all scenarios when trained for 600 epochs. On the undefended traces, DARTS_roneshot shows an improvement of about 1.7 percentage points over Tik-Tok, which exceeds the combined margin of error of 1.4%.³ For the Front_T1 defense, DARTS_roneshot achieves a substantial improvement of 19 percentage points, with a combined margin of error of

³To determine the combined margin of error (MoE) when comparing two models, we sum the squares of their individual standard errors and take the square root of the result: $\text{Combined MoE} = \sqrt{\text{SE}_1^2 + \text{SE}_2^2}$, where SE_1 and SE_2 are the standard errors of the two models. This combined MoE helps in determining if the difference between the accuracies of the two models is statistically significant. If the observed difference in accuracies exceeds the combined MoE, the difference is considered statistically significant, indicating that the performance difference is unlikely due to random chance.

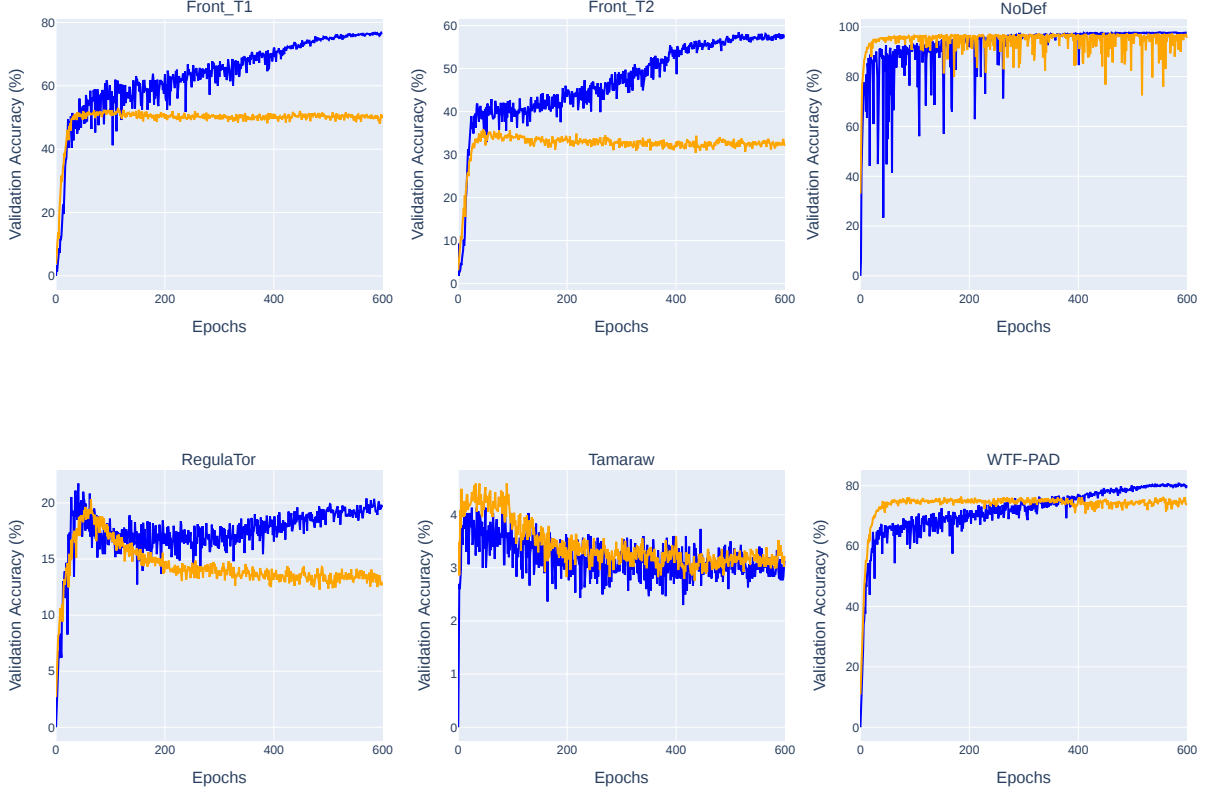


Figure 5.2: Convergence of `DARTS_roneshot` (blue) and Tik-Tok (orange) models on AWF100 datasets over 600 epochs of training.

4.2%. Similarly, on `Front_T2`, `DARTS_roneshot` exhibits a statistically significant difference of 25 percentage points over Tik-Tok, also with a combined margin of error of 4.2%. Despite these gains, the `Tamaraw` defense remains challenging, as both `DARTS_roneshot` and Tik-Tok models perform poorly. For both the `RegulaTor` and `WTF_PAD` defenses, `DARTS_roneshot` surpasses Tik-Tok by 6 percentage points, with combined margins of error of 3.6% and 4.1% respectively, indicating significant enhancements in performance.

5.5.2 Attacking Defended Traces using Early-stopping

Experiments with 600 epochs of training reveal that NAS-generated models, particularly using the `DARTS_roneshot` model, can achieve significantly higher accuracy than those observed with early stopping. This demonstrates the importance of sufficient training

Table 5.6: Validation accuracy for different search strategies and defenses. NAS models were trained using early stopping with a patience of 10. This table includes the Bernoulli error estimates at a 95% confidence interval calculated on the best validation accuracy across 3 cross-validation folds.

Defense	Random One-Shot (%)	DARTS (%)	Gumbel-DARTS (%)	Tik-Tok (%)
NoDef	72 ± 3	87 ± 2	85 ± 2	97 ± 1
Front_T1	58 ± 3	56 ± 3	49 ± 3	49 ± 3
Front_T2	44 ± 3	44 ± 3	43 ± 3	34 ± 3
Tamaraw	4 ± 1	5 ± 1	4 ± 1	5 ± 1
WTF_PAD	68 ± 3	70 ± 3	68 ± 3	71 ± 3
RegulaTor	18 ± 3	23 ± 3	14 ± 2	17 ± 3

epochs to fully realize the potential of NAS-generated architectures. **DARTS_roneshot** model proves to be an effective strategy, especially when given adequate training time, outperforming Tik-Tok in most scenarios. With these results, although we establish the effectiveness of the random one-shot strategy, it remains unclear how it compares to the DARTS and Gumbel-DARTS search strategies in finding DNNs for WF on defended traces. To determine this, we run random one-shot, DARTS, and Gumbel-DARTS on defended AWF traces, incorporating early stopping to save on training time. Note that these results obtained by the NAS models with early stopping represent a lower bound of what can actually be achieved if we let them run for longer durations.

The results from our study on the effectiveness of NAS-generated models in fingerprinting Tor traces defended with WF defenses implemented with early stopping patience of 10 are detailed in Table 5.6. Do note that Tik-Tok did not need early stopping and was trained for the full 30 epochs.

NAS models with early-stopping underperform on NoDef. On the undefended (NoDef) dataset, the random one-shot strategy underperforms significantly, with a relative difference of 25 percentage points (combined margin of error: 4%) compared to Tik-Tok. DARTS also shows a lower performance with a difference of 10 percentage points (combined margin of error: 2.2%), while Gumbel-DARTS is down by 12 percentage points (combined margin of error: 2.2%). However, it is important to note that the accuracy here, with early stopping logic, is significantly lower than the accuracy observed with models trained for

the full 600 epochs (Table 5.2). These poor results can be attributed to the NAS-generated models being under-trained.

NAS models outperform on Front. On both the Front_T1 and Front_T2 defenses, NAS models consistently outperform Tik-Tok. For Front_T1, Random One-Shot achieves $58\% \pm 3\%$ and DARTS reaches $56\% \pm 3\%$, both significantly higher than Tik-Tok’s $49\% \pm 3\%$. The differences of 9% and 7%, respectively, exceed the combined margin of error of 4%. Random One-Shot and DARTS also surpass Gumbel-DARTS, which has an accuracy of $49\% \pm 3\%$, with significant differences of 9% and 7% respectively. Similarly, on Front_T2, all three NAS strategies perform better than Tik-Tok. Random One-Shot achieves $44\% \pm 3\%$, DARTS reaches $44\% \pm 3\%$, and Gumbel-DARTS has $43\% \pm 3\%$, all significantly higher than Tik-Tok’s $34\% \pm 3\%$. The differences are 10%, 10%, and 9%, respectively, all exceeding the combined margin of error of 4%. Among the NAS models, there are no significant differences between Random One-Shot, DARTS, and Gumbel-DARTS for the Front_T2 defense.

All models perform poorly against the Tamaraw defense. On the Tamaraw defense, all NAS models and Tik-Tok show very low accuracies, with no significant differences between them, indicating that Tamaraw is effective in mitigating DNN-based WF attacks. Nevertheless, its substantial timing and bandwidth overheads make this method impractical for deployment in Tor [Per13].

Comparable performance across models on WTF_PAD. On the WTF_PAD defense, the differences in accuracies among the models are all within the combined margin of error of 4%. Similarly, the differences between DARTS and Gumbel-DARTS (2%), DARTS and Tik-Tok (1%), and Gumbel-DARTS and Tik-Tok (3%) are also not significant. These results indicate that while there are small variations in performance among the models, they are not substantial enough to be considered significant, suggesting that all three NAS strategies and Tik-Tok perform similarly on the WTF_PAD dataset.

Significant improvements with DARTS on RegulaTor. On the RegulaTor defense, the DARTS strategy demonstrates significantly higher performance compared to the other models. The differences between DARTS and Random One-Shot (5%), DARTS and Gumbel-DARTS (9%), and DARTS and Tik-Tok (6%) all exceed the combined margin of error of approximately 3.6%, 3.3%, and 3.5% respectively. Random One-Shot also outperforms Gumbel-DARTS with a difference of 4%, which exceeds the combined margin of error of 3.3%. However, the difference between Random One-Shot and Tik-Tok (1%) is not significant as it falls within the combined margin of error of 3.5%. Similarly, the difference between Gumbel-DARTS and Tik-Tok (3%) is not significant. These results suggest that while DARTS has a clear advantage on the RegulaTor defense, Random One-Shot

also shows a notable edge over Gumbel-DARTS. However, the performance differences between Random One-Shot and Tik-Tok, and between Gumbel-DARTS and Tik-Tok, are not substantial enough to be considered significant, indicating that RegulaTor is effective in reducing the success of DNN-based WF attacks, although NAS models, particularly Random One-Shot and DARTS, perform slightly better.

Additionally, our earlier results obtained when training `DARTS_roneshot` models for 600 epochs (Section 5.5.1) prompt us to believe that models generated by DARTS and Gumbel-DARTS can also perform better if trained for a greater number of epochs.

5.6 Generalizability of FAWNS’ DNN Architectures

To further demonstrate the effectiveness of the models generated by FAWNS, we conducted two crucial experiments to evaluate the ability of our NAS-generated models to generalize to unseen data. The first experiment investigates the impact of increasing the number of instances or traces per website on classification accuracy during testing. The second experiment examines the resilience of the trained models to concept drift over time.

5.6.1 Tolerance to Training on a Downsampled Dataset

Increasing the number of testing samples helps to evaluate the generalizing ability of website fingerprinting models by exposing them to a broader spectrum of data variations. When we increase the number of testing samples, we introduce a wider range of variations and potential anomalies that the model must handle. This includes different browsing behaviors and network conditions. A model with strong generalizing ability should be able to maintain good accuracy and performance across diverse samples, showing its capability to adapt to new test traces. Conversely, if the model’s performance degrades significantly with more testing samples, it may indicate overfitting, where the model has learned the training data too specifically and struggles to generalize to new instances. Moreover, increasing the number of testing samples provides a more statistically significant evaluation of the model’s performance. With a larger dataset, the results are less likely to be influenced by outliers or specific peculiarities of a smaller test set.

To determine whether our NAS-generated models maintain stable classification accuracy when tested on larger AWF sets, we searched for and trained six different models using the random one-shot technique on the DARTS search space for 600 epochs. These six models correspond to the one undefended and five defended versions (Front_T1, Front_T2,

Table 5.7: Comparison of test accuracy reported with Bernoulli error estimate at 95% confidence interval between DARTS_roneshot and Tik-Tok on both undefended and defended AWF datasets with increasing number of traces per class.

Model	Defense	Trained & Tested on AWF100 (%)	Tested on AWF200 (%)	Tested on AWF450 (%)
DARTS_roneshot	NoDef	98.0 ± 0.9	99.0 ± 0.5	98.0 ± 0.4
	Front_T1	77 ± 3	78.3 ± 0.8	77 ± 1
	Front_T2	62 ± 3	63 ± 2	62 ± 1
	WTF_PAD	86 ± 2	87 ± 1	87 ± 1
	Tamaraw	4 ± 1	2.8 ± 0.7	3.3 ± 0.5
	RegulaTor	22 ± 3	22 ± 2	20 ± 1
Tik-Tok	NoDef	94 ± 1	96.2 ± 0.8	95.1 ± 0.6
	Front_T1	47 ± 3	49 ± 2	48 ± 2
	Front_T2	34 ± 3	36 ± 2	34 ± 1
	WTF_PAD	70 ± 3	73 ± 2	73 ± 1
	Tamaraw	4 ± 1	4.6 ± 0.9	4.2 ± 0.6
	RegulaTor	17 ± 2	19 ± 2	17 ± 2

WTF-PAD, Tamaraw, RegulaTor) of the AWF100 dataset. We then tested these models on the test sets of the AWF100, AWF200, and AWF450 datasets, which contain 1,000, 2,000, and 4,500 test samples belonging to 100 classes, respectively. Table 5.7 highlights the comparative performance of models generated using the random one-shot technique and the Tik-Tok model across AWF datasets with increasing traces per class.

DARTS_roneshot model consistently outperforms Tik-Tok on the undefended dataset. For defenses like Front_T1, Front_T2, and WTF_PAD, the DARTS_roneshot model maintains higher accuracies. These results suggest that NAS-generated models exhibit a strong robustness as the number of traces per class increases for these defenses. However, both models face difficulties with the Tamaraw and RegulaTor defenses perhaps because of their robust defense design. Tamaraw employs constant-rate padding to obscure traffic patterns, making fingerprinting challenging. However, as stated earlier, this approach incurs significant timing and bandwidth overheads, making these defenses impractical for deployment in Tor [Per13]. RegulaTor, on the other hand, transmits packets at a predefined rate to prevent information leakage about traffic volume and duration, complicating the discrimination of statistical differences between website traces from different classes.

Table 5.8: DARTS_oneshot and Tik-Tok accuracy for different time drifts.

AWF with time drift	DARTS_oneshot (%)	Deep Fingerprinting (%)
Baseline	97 ± 1	98 ± 1
Baseline + 1 week	92 ± 2	98.1 ± 0.8
Baseline + 2 weeks	81 ± 2	84 ± 2
Baseline + 4 weeks	74 ± 3	78 ± 3
Baseline + 6 weeks	76 ± 3	74 ± 3

5.6.2 Tolerance to Concept Drift

The challenge of correctly recognizing traffic traces collected over time was first addressed by Juarez et al. [JAA⁺14], who demonstrated that classification accuracy significantly drops when the model is tested on traffic captured 10 days after training. This decline is attributed to dynamic changes on the content of websites, which affect the shape of their identifying fingerprints. Additionally, classifiers might learn temporary features instead of stable fingerprints when trained and evaluated simultaneously. This phenomenon is known as concept drift, where statistical properties of the target class change over time, making recognition less accurate. A model resilient to concept drift captures salient traffic features highly correlated with website fingerprints, maintaining performance over time.

To evaluate NAS-generated models’ tolerance to concept drift, we trained the models on a closed world version of the AWF dataset collected by Rimmer et al. [RPJ⁺18] and tested them on data collected by them periodically over six weeks from the same closed-world websites. These new website traces are directional only. Thus, we will refer to the attack we compare our model with as the DF attack, rather than Tik-Tok, which utilizes directional-timing features. More specifically, we analyzed the performance of two models, DARTS_oneshot and DF, trained on the baseline AWF closed-world set and evaluated their accuracy on traffic re-collected 10 days, 4 weeks, and 6 weeks from the same closed-world of 100 classes with 100 traces each. The results in Table 5.8 indicate a decrease in classification accuracy over time. Despite a significant six-week gap between training and the last evaluation, the models still correctly identify at least 71% of the website visits in the test sets.

High accuracy initially, declines over time due to website changes. Both models initially perform well on the held-out test set, indicating they have effectively captured the identifying features of the websites from the training data. However, their accuracy declines

over time due to changes in website content and structure. After 10 days, `DARTS_roneshot` drops notably from $97\% \pm 1\%$ to $92\% \pm 2\%$, highlighting its higher sensitivity to short-term changes. Over two weeks, both models experience significant drops in accuracy, with `DARTS_roneshot` decreasing to $81\% \pm 2\%$ (a decrease of 11%, combined margin of error 2.8%) and deep fingerprinting dropping to $84\% \pm 2\%$ (a decrease of 14%, combined margin of error 2.8%), reflecting the impact of concept drift. By four weeks, accuracy continues to decrease, with `DARTS_roneshot` at $74\% \pm 3\%$ and DF at $78\% \pm 3\%$. While `DARTS_roneshot` maintains relatively stable performance with $76\% \pm 3\%$ from 4 to 6 weeks, Deep Fingerprinting shows a further decline to $74\% \pm 3\%$.

These findings highlight the impact of concept drift over time and the varying sensitivities of the `DARTS_roneshot` and DF models. Although both models experience a decrease in accuracy as the time gap between training and testing increases, they handle this drop differently. This variation in sensitivity emphasizes the need to consider temporal changes when evaluating website fingerprinting models. Despite the overall decline in accuracy, the models maintain a substantial level of performance over extended periods, indicating a strong generalizing ability. This implies and agrees with Rimmer et al. [RPJ⁺18] that DNN-based attacks are capable of capturing stable and salient features from the traffic data, which remain relevant even as the underlying web content evolves over time.

Chapter 6

Conclusions

Users rely on low-latency anonymity networks like Tor to protect online privacy by creating multi-hop encrypted pathways. However, because Tor’s design retains the timing and volume characteristics of packet exchanges, it is vulnerable to website fingerprinting (WF) attacks. These attacks use machine learning techniques to match users’ traffic patterns to pre-recorded fingerprints and identify visited websites. Recent WF research has increasingly utilized deep neural networks (DNNs) for more precise attacks. Traditionally, the search for effective DNN architectures relied on trial-and-error, adapting designs from related fields. As this method lacks a systematic approach to identify performance-enhancing elements and may overlook innovative designs, recent studies have applied automated machine learning, specifically neural architecture search (NAS) and have demonstrated its effectiveness in generating WF attacks.

NAS is a promising and rapidly advancing research field, with many algorithms successfully discovering efficient models for various domains, including computer vision and natural language processing. To provide a unified framework for WF research to explore these novel NAS techniques for fingerprinting websites, we devised FAWNS. This neural architecture search framework is specifically designed for website fingerprinting and integrates with Microsoft’s NNI to abstract the technical aspects of NAS from users, offering a modular and adaptable framework compatible with standard website fingerprinting datasets.

Users can leverage FAWNS’ website fingerprinting capabilities by simply specifying their choice of search space and search strategy from those already integrated. For users interested in tweaking NAS search spaces or experimenting with novel NAS strategies, FAWNS, in conjunction with NNI’s APIs, provides the flexibility to do so. Additionally,

we envision that the FAWNS framework can be easily adapted for other traffic analysis tasks, such as traffic correlation.

Through the FAWNS framework, we devised DNNs for WF and evaluated their effectiveness at deanonymizing Tor website visits. The results of our evaluation reveal that FAWNS-generated DNNs achieve comparable accuracy to the Tik-Tok attack on undefended Tor traffic and substantially higher accuracy on defended Tor traffic. These findings agree with our thesis statement, highlighting that DNN architectures specifically searched and optimized on WF datasets outperform existing DNN-based WF attacks that directly borrow or adapt DNN architectures from related domains. Our results suggest that NAS techniques have significant potential to enhance the effectiveness of WF attacks. Additionally, our research highlights several directions for future work in this area, which we propose in the following paragraphs.

Limitations and Future Work

Our work on FAWNS opens up several interesting directions for future research.

Explainability of FAWNS-generated models. NAS often results in complex models that are difficult to interpret. Implementing explainability techniques, such as integrating feature importance scores [BGG⁺23] or using interpretable surrogate models [NM19], could make the decision-making processes of FAWNS-generated models more transparent.

Generalizability across datasets. Another promising direction is to evaluate and improve the generalizability of FAWNS-generated models across different datasets. This involves training models on one dataset, such as AWF, and testing them on another, like DS-19. This cross-dataset evaluation can reveal how well the models generalize to new, unseen data and help identify potential drawbacks. Enhancing generalizability would make FAWNS a more robust and versatile tool for various traffic analysis tasks.

Expanding search spaces. Since this thesis focused solely on the DARTS search space, we plan to explore other search spaces like ENAS [PGZ⁺18] and PNAS [LZN⁺18] to understand how the effectiveness of discovered architectures is influenced by the search spaces and the operations they support.

Adapting FAWNS for new trace representations. FAWNS currently works with directional-timing vectors. We recognize that recent WF research is shifting focus from creating more accurate DNNs to developing more robust trace representations [SJG⁺23]. We aim to modify FAWNS’ input pipeline to accommodate other trace representations.

Evaluating FAWNS in open-world setting. Evaluating FAWNS-generated DNNs for open-world website fingerprinting will help us better understand and improve the generalizability and robustness of these models in more complex and realistic scenarios. This effort will focus on running experiments with open-world datasets, collecting and analyzing results as the FAWNS framework already has all the necessary support implemented to run open-world experiments.

More rigorous hyperparameter tuning. We plan to delve deeper into meticulously tuning the hyperparameters of more complex search algorithms like DARTS. By optimizing these parameters such as the learning rate, weight decay, and gradient clipping value, we aim to enhance the test accuracy of the models. This careful tuning process is crucial as it can significantly impact the performance of NAS-generated models.

FAWNS for autotransformers. Recent research in WF has employed transformer models to address multi-tab website fingerprinting [JLLS23, DYL⁺23]. Exploring the potential of FAWNS to implement autotransformers [CPFL21, SLL19] for generating transformer models for multi-tab WF would be interesting.

References

- [ACMP18] Giuseppe Aceto, Domenico Ciuonzo, Antonio Montieri, and Antonio Pescapé. Mobile encrypted traffic classification using deep learning. In *Proceedings of the Network Traffic Measurement and Analysis Conference*, pages 1–8, 2018.
- [AG16] Kota Abe and Shigeki Goto. Fingerprinting attack on Tor anonymity using deep learning. *Proceedings of the Asia-Pacific Advanced Network*, 42:15–20, 2016.
- [AMD121] Mohamed S Abdelfattah, Abhinav Mehrotra, Łukasz Dudziak, and Nicholas D Lane. Zero-cost proxies for lightweight NAS. In *Proceedings of the International Conference on Learning Representations*, pages 1–17, 2021.
- [BBBK11] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in Neural Information Processing Systems*, 24:2546–2554, 2011.
- [BCB15] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *Proceedings of the International Conference on Learning Representations*, 2015.
- [BGG⁺23] Francesco Bodria, Fosca Giannotti, Riccardo Guidotti, Francesca Naretto, Dino Pedreschi, and Salvatore Rinzivillo. Benchmarking and survey of explanation methods for black box models. *Data Mining and Knowledge Discovery*, 37(5):1719–1778, 2023.
- [BGRN17] Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. Accelerating neural architecture search using performance prediction. *arXiv preprint arXiv:1705.10823*, 2017.

- [BKZ⁺18] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In *Proceedings of the International Conference on Machine Learning*, pages 550–559, 2018.
- [BLC⁺20] Gabriel Bender, Hanxiao Liu, Bo Chen, Grace Chu, Shuyang Cheng, Pieter-Jan Kindermans, and Quoc V Le. Can weight sharing outperform random architecture search? An investigation with TuNAS. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 14323–14332, 2020.
- [BLKD19] Sanjit Bhat, David Lu, Albert Kwon, and Srinivas Devadas. Var-CNN: A data-efficient website fingerprinting attack based on deep learning. In *Proceedings on Privacy Enhancing Technologies*, volume 4, pages 292–310, 2019.
- [BLRW18] Andrew Brock, Theo Lim, J.M. Ritchie, and Nick Weston. SMASH: One-Shot Model Architecture Search through HyperNetworks. In *Proceedings of the International Conference on Learning Representations*, 2018.
- [CCZ⁺18] Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Efficient architecture search by network transformation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.
- [Che17] Giovanni Cherubin. Bayes, not naïve: Security bounds on website fingerprinting defenses. In *Proceedings on Privacy Enhancing Technologies*, volume 2017, issue 4, pages 215–231, 2017.
- [CHJ17] Giovanni Cherubin, Jamie Hayes, and Marc Juárez. Website Fingerprinting Defenses at the Application Layer. In *Proceedings on Privacy Enhancing Technologies*, volume 2017, issue 2, pages 186–203, 2017.
- [CKM21] Aristeidis Christoforidis, George Kyriakides, and Konstantinos Margaritis. A novel evolutionary algorithm for hierarchical neural architecture search. *arXiv preprint arXiv:2107.08484*, 2021.
- [CLH17] Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. A downsampled variant of ImageNet as an alternative to the CIFAR datasets. *arXiv preprint arXiv:1707.08819*, 2017.

- [CNJ14] Xiang Cai, Rishab Nithyanand, and Rob Johnson. CS-BuFLO: A congestion sensitive website fingerprinting defense. In *Proceedings of the 13th ACM Workshop on Privacy in the Electronic Society*, pages 121–130, 2014.
- [CNW⁺14] Xiang Cai, Rishab Nithyanand, Tao Wang, Rob Johnson, and Ian Goldberg. A systematic approach to developing and evaluating website fingerprinting defenses. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 227–238, 2014.
- [CPFL21] Minghao Chen, Houwen Peng, Jianlong Fu, and Haibin Ling. Autoformer: Searching transformers for visual recognition. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 12270–12280, 2021.
- [CYZ⁺18] Han Cai, Jiacheng Yang, Weinan Zhang, Song Han, and Yong Yu. Path-level network transformation for efficient architecture search. In *Proceedings of the International Conference on Machine Learning*, pages 678–687, 2018.
- [CZH19] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *Proceedings of the International Conference on Learning Representations*, 2019.
- [CZJJ12] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *Proceedings of the 2012 ACM SIGSAC Conference on Computer and Communications Security*, pages 605–616, 2012.
- [DAAW22] Priyanka Dodia, Masha'al AlSabah, Omar Alrawi, and Tao Wang. Exposing the rat in the tunnel: Using traffic analysis for Tor-based malware detection. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 875–889, 2022.
- [DCRS12] Kevin P Dyer, Scott E Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, pages 332–346, 2012.
- [DICMH⁺20] Wladimir De la Cadena, Asya Mitseva, Jens Hiller, Jan Pennekamp, Sebastian Reuter, Julian Filter, Thomas Engel, Klaus Wehrle, and Andriy Panchenko. Trafficsliver: Fighting website fingerprinting attacks with traffic splitting. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1971–1985, 2020.

- [DMS⁺04] Roger Dingledine, Nick Mathewson, Paul F Syverson, et al. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, volume 4, pages 303–320, 2004.
- [DYL⁺23] Xinhao Deng, Qilei Yin, Zhuotao Liu, Xiyuan Zhao, Qi Li, Mingwei Xu, Ke Xu, and Jianping Wu. Robust multi-tab website fingerprinting attacks in the wild. In *Proceedings of the 44th IEEE Symposium on Security and Privacy*, pages 1005–1022, 2023.
- [EMH17] Thomas Elsken, Jan-Hendrik Metzen, and Frank Hutter. Simple and efficient architecture search for convolutional neural networks. *arXiv preprint arXiv:1711.04528*, 2017.
- [EMH18] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via lamarckian evolution. *arXiv preprint arXiv:1804.09081*, 2018.
- [EMH19] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.
- [FKE⁺15] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. *Advances in Neural Information Processing Systems*, Vol. 28, 2015.
- [FMM17] Saya Fujino, Naoki Mori, and Keinosuke Matsumoto. Deep convolutional networks for human sketches by means of the evolutionary deep learning. In *Proceedings of the joint 17th world congress of International Fuzzy Systems Association and 9th International Conference on Soft Computing and Intelligent Systems*, pages 1–5, 2017.
- [GD91] David E Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of genetic algorithms*, volume 1, pages 69–93. Elsevier, 1991.
- [GLL18] Spencer Gibb, Hung Manh La, and Sushil Louis. A genetic algorithm for convolutional network structure optimization for concrete crack detection. In *Proceedings of the 2018 IEEE Congress on Evolutionary Computation*, pages 1–8, 2018.

- [Gu22] Sonia Gu. Leveraging Automated Machine Learning for Website Fingerprinting. *Princeton University Undergraduate Senior Theses*, 2022.
- [GW20] Jiajun Gong and Tao Wang. Zero-delay lightweight defenses against website fingerprinting. In *Proceedings of the 29th USENIX Security Symposium*, pages 717–734, 2020.
- [GZZW22] Jiajun Gong, Wuqi Zhang, Charles Zhang, and Tao Wang. Surakav: Generating realistic traces for a strong website fingerprinting defense. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy*, pages 1558–1573, 2022.
- [GZZW23] Jiajun Gong, Wuqi Zhang, Charles Zhang, and Tao Wang. WFDefProxy: Modularly implementing and empirically evaluating website fingerprinting defenses. *IEEE Transactions on Information Forensics and Security*, 2023.
- [HCOH23] James K Holland, Jason Carpenter, Se Eun Oh, and Nicholas Hopper. DeTorrent: An Adversarial Padding-only Traffic Analysis Defense. In *Proceedings on Privacy Enhancing Technologies*, volume 2024, issue 1, pages 98–115, 2023.
- [HD16] Jamie Hayes and George Danezis. k-fingerprinting: A Robust Scalable Website Fingerprinting Technique. In *Proceedings of the 25th USENIX Security Symposium*, pages 1187–1203, 2016.
- [HGS⁺20] Sébastien Henri, Ginés García, Pablo Serrano, Albert Banchs, and Patrick Thiran. Protecting against website fingerprinting with multihoming. In *Proceedings on Privacy Enhancing Technologies*, volume 2020, issue 2, pages 89–110. Sciendo, 2020.
- [HH22] James K Holland and Nicholas Hopper. RegulaTor: A straightforward website fingerprinting defense. In *Proceedings on Privacy Enhancing Technologies*, volume 2022, issue 2, pages 344–362. Sciendo, 2022.
- [HHLB11] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th Conference on Learning and Intelligent Optimization*, pages 507–523. Springer, 2011.
- [HKV19] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. *Automated machine learning: methods, systems, challenges*. Springer Nature, 2019.

- [HLVDMW17] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4700–4708, 2017.
- [HSFM21] Jordan Holland, Paul Schmitt, Nick Feamster, and Prateek Mittal. New directions in automated traffic analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3366–3383, 2021.
- [HWF09] Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. Website fingerprinting: attacking popular privacy enhancing technologies with the multinomial naïve-bayes classifier. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 31–42, 2009.
- [HZRS16] 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, 2016.
- [JAA⁺14] Marc Juarez, Sadia Afroz, Gunes Acar, Claudia Diaz, and Rachel Greenstadt. A critical evaluation of website fingerprinting attacks. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 263–274, 2014.
- [JCSH23] Haifeng Jin, François Chollet, Qingquan Song, and Xia Hu. AutoKeras: An AutoML Library for Deep Learning. *Journal of Machine Learning Research*, Vol. 24(6):1–6, 2023.
- [JJH⁺17] Aaron Johnson, Rob Jansen, Nicholas Hopper, Aaron Segal, and Paul Syverson. PeerFlow: Secure load balancing in Tor. In *Proceedings on Privacy Enhancing Technologies*, volume 2017, issue 2, pages 74–94, 2017.
- [JLLS23] Zhaoxin Jin, Tianbo Lu, Shuang Luo, and Jiaze Shang. Transformer-based Model for Multi-tab Website Fingerprinting Attack. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1050–1064, 2023.
- [JSH19] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-Keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data mining*, pages 1946–1956, 2019.

- [KFB⁺17] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast bayesian optimization of machine learning hyperparameters on large datasets. In *Proceedings of the Conference on Artificial Intelligence and Statistics*, pages 528–536, 2017.
- [KNS⁺18] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric P Xing. Neural architecture search with bayesian optimisation and optimal transport. *Advances in Neural Information Processing Systems*, 31, 2018.
- [KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 25, 2012.
- [LCC10] Liming Lu, Ee-Chien Chang, and Mun Choon Chan. Website fingerprinting and identification using ordered feature sequences. In *Proceedings of the 15th European Symposium on Research in Computer Security*, pages 199–214. Springer, 2010.
- [LDH⁺20] Chenxi Liu, Piotr Dollár, Kaiming He, Ross Girshick, Alan Yuille, and Saining Xie. Are labels necessary for neural architecture search? In *Proceedings of the 16th European Conference on Computer Vision*, pages 798–813. Springer, 2020.
- [Liu18] Liu, Hanxiao and Simonyan, Karen and Vinyals, Oriol and Fernando, Chrisantha and Kavukcuoglu, Koray. Hierarchical representations for efficient architecture search. In *Proceedings of the International Conference on Learning Representations*, 2018.
- [LL06] Marc Liberatore and Brian Neil Levine. Inferring the source of encrypted HTTP connections. In *Proceedings of the 13th ACM SIGSAC Conference on Computer and Communications Security*, pages 255–263, 2006.
- [LSY19] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable Architecture Search. In *Proceedings of the International Conference on Learning Representations*, 2019.
- [LT20] Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pages 367–377, 2020.

- [LZC⁺11] Xiapu Luo, Peng Zhou, Edmond WW Chan, Wenke Lee, Rocky KC Chang, Roberto Perdisci, et al. HTTPOS: Sealing Information Leaks with Browser-side Obfuscation of Encrypted Flows. In *Proceedings of the 18th Annual Network and Distributed System Security*, volume 11, 2011.
- [LZN⁺18] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision*, pages 19–34, 2018.
- [MAS⁺23] Navid Malekghaini, Elham Akbari, Mohammad A Salahuddin, Noura Limam, Raouf Boutaba, Bertrand Mathieu, Stephanie Moteau, and Stephane Tuffin. Deep learning for encrypted traffic classification in the face of data drift: An empirical study. *Computer Networks*, 225:109648, 2023.
- [MCRG23] Pedro Mendes, Maria Casimiro, Paolo Romano, and David Garlan. HyperJump: Accelerating HyperBand via Risk Modelling. *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37(8):9143–9152, 2023.
- [MCY19] Lizheng Ma, Jiaxu Cui, and Bo Yang. Deep neural architecture search with deep graph bayesian optimization. In *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence*, pages 500–507, 2019.
- [MD05] Steven J Murdoch and George Danezis. Low-cost traffic analysis of Tor. In *Proceedings of the 26th IEEE Symposium on Security and Privacy*, pages 183–195, 2005.
- [MHO⁺23] Nate Mathews, James K Holland, Se Eun Oh, Mohammad Saidur Rahman, Nicholas Hopper, and Matthew Wright. SoK: A critical evaluation of efficient website fingerprinting defenses. In *Proceedings of the 44th IEEE Symposium on Security and Privacy*, pages 969–986, 2023.
- [Mic21] Microsoft. Neural Network Intelligence, 1 2021.
- [MKS⁺15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

- [NBH21] Milad Nasr, Alireza Bahramali, and Amir Houmansadr. Defeating DNN-Based Traffic Analysis Systems in Real-Time With Blind Adversarial Perturbations. In *Proceedings of the 30th USENIX Security Symposium*, pages 2705–2722, 2021.
- [NCJ14] Rishab Nithyanand, Xiang Cai, and Rob Johnson. Glove: A bespoke website fingerprinting defense. In *Proceedings of the 13th ACM Workshop on Privacy in the Electronic Society*, pages 131–134, 2014.
- [NG17] Renato Negrinho and Geoff Gordon. DeepArchitect: Automatically designing and training deep architectures. *arXiv preprint arXiv:1704.08792*, 2017.
- [NLYO21] Vu Nguyen, Tam Le, Makoto Yamada, and Michael A Osborne. Optimal transport kernels for sequential and parallel neural architecture search. In *Proceedings of the International Conference on Machine Learning*, pages 8084–8095, 2021.
- [NM19] Caio Nóbrega and Leandro Marinho. Towards explaining recommendations through local surrogate models. In *Proceedings of the 34th ACM Special Interest Group on Applied Computing symposium on applied computing*, pages 1671–1678, 2019.
- [Per11] Mike Perry. Experimental Defense Against Website Traffic Fingerprinting. <https://blog.torproject.org/experimental-defense-website-traffic-fingerprinting/>, September 5 2011. Accessed: July 31, 2024.
- [Per13] Mike Perry. A Critique of Website Traffic Fingerprinting Attacks. <https://blog.torproject.org/critique-website-traffic-fingerprinting-attacks/>, November 7 2013. Accessed: July 31, 2024.
- [PGZ⁺18] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. In *Proceedings of the International Conference on Machine Learning*, pages 4095–4104, 2018.
- [PLP⁺16] Andriy Panchenko, Fabian Lanze, Jan Pennekamp, Thomas Engel, Andreas Zinnen, Martin Henze, and Klaus Wehrle. Website Fingerprinting at Internet Scale. In *Proceedings of the 23rd Network and Distributed System Security Symposium*, 2016.

- [PNZE11] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website fingerprinting in onion routing based anonymization networks. In *Proceedings of the 10th ACM Workshop on Privacy in the Electronic Society*, pages 103–114, 2011.
- [RAHL19a] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Aging evolution for image classifier architecture search. In *Proceedings of the AAAI Conference on artificial intelligence*, volume 2, page 2, 2019.
- [RAHL19b] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI Conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.
- [REC20] Robin Ru, Pedro Esperanca, and Fabio Maria Carlucci. Neural architecture generator optimization. *Advances in Neural Information Processing Systems*, 33:12057–12069, 2020.
- [RIMW20] Mohammad Saidur Rahman, Mohsen Imani, Nate Mathews, and Matthew Wright. Mockingbird: Defending against deep-learning-based website fingerprinting attacks with adversarial traces. *IEEE Transactions on Information Forensics and Security*, 16:1594–1609, 2020.
- [RJX⁺19] Ilija Radosavovic, Justin Johnson, Saining Xie, Wan-Yen Lo, and Piotr Dollár. On network design spaces for visual recognition. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1882–1890, 2019.
- [RLSL20] Esteban Real, Chen Liang, David So, and Quoc Le. AutoML-Zero: Evolving machine learning algorithms from scratch. In *Proceedings of the International Conference on Machine Learning*, pages 8007–8019, 2020.
- [RMS⁺17] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In *Proceedings of the International Conference on Machine Learning*, pages 2902–2911, 2017.
- [RPJ⁺18] Vera Rimmer, Davy Preuveneers, Marc Juarez, Tom Van Goethem, and Wouter Joosen. Automated website fingerprinting through deep learning. In *Proceedings of the 25th Network and Distributed Systems Security Symposium*, 2018.

- [RSM⁺20] Mohammad Saidur Rahman, Payap Sirinam, Nate Mathews, Kantha Girish Gangadhara, and Matthew Wright. Tik-Tok: The utility of packet timing in website fingerprinting attacks. In *Proceedings on Privacy Enhancing Technologies*, volume 2020, page 5–24, 2020.
- [RWDO21] Binxin Ru, Xingchen Wan, Xiaowen Dong, and Michael Osborne. Interpretable Neural Architecture Search via Bayesian Optimisation with Weisfeiler-Lehman Kernels. In *Proceedings of the International Conference on Learning Representations*, 2021.
- [RXC⁺21] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. A Comprehensive Survey of Neural Architecture Search: Challenges and Solutions. *ACM Computing Surveys*, Vol. 54(4), 2021.
- [SANM⁺23] Prabhjot Singh, Shreya Arun Naik, Navid Malekghaini, Diogo Barradas, and Noura Limam. A First Look at Generating Website Fingerprinting Attacks via Neural Architecture Search. In *Proceedings of the 22nd ACM Workshop on Privacy in the Electronic Society*, pages 173–178, 2023.
- [SBZZ21] Shawn Shan, Arjun Nitin Bhagoji, Haitao Zheng, and Ben Y Zhao. Patch-based defenses against web fingerprinting attacks. In *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security*, pages 97–109, 2021.
- [SDS⁺14] Kevin Swersky, David Duvenaud, Jasper Snoek, Frank Hutter, and Michael A Osborne. Raiders of the lost architecture: Kernels for Bayesian optimization in conditional parameter spaces. *arXiv preprint arXiv:1409.4011*, 2014.
- [SHM⁺16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.
- [SHZ⁺18] 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.

- [SIJW18] Payap Sirinam, Mohsen Imani, Marc Juarez, and Matthew Wright. Deep fingerprinting: Undermining website fingerprinting defenses with deep learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1928–1943, 2018.
- [SJG⁺23] Meng Shen, Kexin Ji, Zhenbo Gao, Qi Li, Liehuang Zhu, and Ke Xu. Subverting Website Fingerprinting Defenses with Robust Traffic Representation. In *Proceedings of the 32nd USENIX Security Symposium*, pages 607–624, 2023.
- [SKFH16] Jost Tobias Springenberg, Aaron Klein, Stefan Falkner, and Frank Hutter. Bayesian optimization with robust Bayesian neural networks. *Advances in Neural Information Processing Systems*, 29, 2016.
- [SLA12] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in Neural Information Processing Systems*, 25, 2012.
- [SLJ⁺15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.
- [SLL19] David So, Quoc Le, and Chen Liang. The evolved transformer. In *Proceedings of the International Conference on Machine Learning*, pages 5877–5886, 2019.
- [SPX⁺20] Han Shi, Renjie Pi, Hang Xu, Zhenguo Li, James Kwok, and Tong Zhang. Bridging the gap between sample-based and one-shot neural architecture search with BONAS. *Advances in Neural Information Processing Systems*, 33:1808–1819, 2020.
- [SV16] Shreyas Saxena and Jakob Verbeek. Convolutional neural fabrics. *Advances in Neural Information Processing Systems*, 29, 2016.
- [SW06] Vitaly Shmatikov and Ming-Hsiu Wang. Timing analysis in low-latency mix networks: Attacks and defenses. In *Proceedings of the 11th European Symposium on Research in Computer Security*, pages 18–33. Springer, 2006.

- [SXZY19] Yanan Sun, Bing Xue, Mengjie Zhang, and Gary G Yen. Evolving deep convolutional neural networks for image classification. *IEEE Transactions on Evolutionary Computation*, 24(2):394–407, 2019.
- [SZ15] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 3rd International Conference on Learning Representations*, 2015.
- [VRB23] Alexander Veicht, Cedric Renggli, and Diogo Barradas. DeepSE-WF: Unified Security Estimation for Website Fingerprinting Defenses. In *Proceedings on Privacy Enhancing Technologies*, volume 2, pages 188–205, 2023.
- [VSP⁺17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 2017.
- [Wan16] Tao Wang. *Website fingerprinting: Attacks and defenses*. PhD thesis, University of Waterloo, 2016.
- [WCM09] Charles V Wright, Scott E Coull, and Fabian Monroe. Traffic Morphing: An Efficient Defense Against Statistical Traffic Analysis. In *Proceedings of the 16th Annual Network and Distributed System Security*, volume 9, 2009.
- [WCN⁺14] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. Effective attacks and provable defenses for website fingerprinting. In *Proceedings of the 23rd USENIX Security Symposium*, pages 143–157, 2014.
- [WG13] Tao Wang and Ian Goldberg. Improved website fingerprinting on Tor. In *Proceedings of the 12th ACM Workshop on Privacy in the Electronic Society*, pages 201–212, 2013.
- [WG17] Tao Wang and Ian Goldberg. Walkie-Talkie: An efficient defense against passive website fingerprinting attacks. In *Proceedings of the 26th USENIX Security Symposium*, pages 1375–1390, 2017.
- [Wis18] Martin Wistuba. Finding competitive network architectures within a day using UCT. In *Proceedings of the 5th IEEE International Conference on Data Science and Advanced Analytics*, pages 263–272, 2018.

- [WREC22] Xingchen Wan, Binxin Ru, Pedro M Esparana, and Fabio Maria Carlucci. Approximate neural architecture search via operation distribution learning. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 2377–2386, 2022.
- [WRP19] Martin Wistuba, Ambrish Rawat, and Tejaswini Pedapati. A survey on neural architecture search. *Journal of Machine Learning Research*, 2019.
- [WSS⁺23] Colin White, Mahmoud Safari, Rhea Sukthanker, Binxin Ru, Thomas Elsken, Arber Zela, Debadeepta Dey, and Frank Hutter. Neural architecture search: Insights from 1000 papers. *arXiv preprint arXiv:2301.08727*, 2023.
- [WWRC16] Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen. Network morphism. In *Proceedings of the International Conference on Machine Learning*, pages 564–572, 2016.
- [WZJ⁺19] Linnan Wang, Yiyang Zhao, Yuu Jinnai, Yuandong Tian, and Rodrigo Fonseca. AlphaX: eXploring neural architectures with deep neural networks and monte carlo tree search. *arXiv preprint arXiv:1903.11059*, 2019.
- [WZZ⁺17] Wei Wang, Ming Zhu, Xuwen Zeng, Xiaozhou Ye, and Yiqiang Sheng. Malware traffic classification using convolutional neural network for representation learning. In *Proceedings of the International Conference on Information Networking*, pages 712–717, 2017.
- [XZLL19] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. SNAS: Stochastic Neural Architecture Search. In *Proceedings of the International Conference on Learning Representations*, 2019.
- [YCBL14] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *Advances in Neural Information Processing Systems*, 27, 2014.
- [YKC⁺19] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. NAS-Bench-101: Towards reproducible neural architecture search. In *Proceedings of the International Conference on Machine Learning*, pages 7105–7114, 2019.
- [YSJ⁺20] Kaicheng Yu, Christian Sciuto, Martin Jaggi, Claudiu Musat, and Mathieu Salzmann. Evaluating the search phase of neural architecture search. In

Proceedings of the International Conference on Learning Representations, 2020.

- [YWSH21] Shen Yan, Colin White, Yash Savani, and Frank Hutter. NAS-Bench-x11 and the power of learning curves. *Advances in Neural Information Processing Systems*, 34:22534–22549, 2021.
- [ZKFH18] Arber Zela, Aaron Klein, Stefan Falkner, and Frank Hutter. Towards automated deep learning: Efficient joint neural architecture and hyperparameter search. In *Proceedings of the International Conference on Machine Learning AutoML Workshop*, July 2018.
- [ZLJ⁺20] Yuge Zhang, Zejun Lin, Junyang Jiang, Quanlu Zhang, Yujing Wang, Hui Xue, Chen Zhang, and Yaming Yang. Deeper insights into weight sharing in neural architecture search. *arXiv preprint arXiv:2001.01431*, 2020.
- [ZVSL18] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8697–8710, 2018.