Deep Learning Approaches to Universal and Practical Steganalysis

by

Ajinkya Kishore Nene

S.B., Massachusetts Institute of Technology (2020)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2020

Author .....
Department of Electrical Engineering and Computer Science
May 12, 2020

Certified by .....
Kalyan Veeramachaneni
Principal Research Scientist
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Deep Learning Approaches to Universal and Practical Steganalysis

by

Ajinkya Kishore Nene

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Steganography is the process of hiding data inside of files while steganalysis is the process of detecting the presence of hidden data inside of files. As a concealment system, steganography is effective at safeguarding the privacy and security of information. Due to its effectiveness as a concealment system, bad actors have increasingly begun using steganography to transmit exploits or other malicious information. Steganography thus poses a significant security risk, demanding serious attention and emphasizing a need for universal and practical steganalysis models that can defend against steganography-based attack vectors. In this thesis, we provide a comprehensive review of steganography-enabled exploits and design a robust framework for universal and practical deep-learning steganalysis. As part of our framework, we provide new and practical steganalysis architectures, propose several data augmentation techniques which includes a novel adversarial-attack system, and develop a python library, StegBench, to enable dynamic and robust steganalysis evaluation. Altogether, our framework enables the development of practical and universal steganalysis models that can be used robustly in security applications to neutralize steganography-based threat models.

Thesis Supervisor: Kalyan Veeramachaneni
Title: Principal Research Scientist

Acknowledgments

First and foremost, I would like to thank my advisor, Kalyan Veeramachaneni, for the opportunity to work with his group and for providing his feedback and advice. Kalyan's advising has helped me grow as both a researcher and a student.

I would like to acknowledge the following people. Arash Akhgari for his amazing help with figures in this thesis. Cara Giaimo, Max Suechting, and Kevin Zhang for help with editing. Carles Sala for useful technical feedback on the StegBench system. Ivan Ramirez Diaz for providing helpful technical feedback on the StegAttack system. Without any of these people, this thesis would not have been possible. I would also like to acknowledge the generous funding support from Accenture under their 'Self Healing Applications' program.

Next, I would like to thank all my friends at MIT, especially the brothers of PKT, for the constant support and friendship that they have given me these last few years. During this COVID-era, I have found it resoundingly true that it is the people of MIT that make it such a special place. Each of the friends I have made in these last few years has helped make my four years at the institute some of the best years of my life.

Last but not least, I want to acknowledge my parents and my sister for helping me to get to this stage in life. They have always stood by me and continue to inspire me every day. This thesis is a testament to all the support and love they have shown me my entire life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Steganography is the process of hiding data inside an ordinary (non-secret) file in order to avoid detection. While encryption aims to hide the contents of data, steganography aims to hide the presence of data. By hiding the presence of data, steganography is also able to conceal communication behaviors and thereby provide behavioral security. The behavioral security provided by steganography thus plays a critical role in safeguarding information privacy.

However, while steganographic concealment systems are regularly used for benign tasks, they may also be used by bad actors to transmit malicious information using ordinary files such as images, thereby posing a security risk. Since current network defenses do not check images for steganographic content, they are unable to effectively block the transmission of malicious steganographic content [25]. Thus, hackers can leverage steganography to transmit exploits or other compromising information across networks in an undetectable fashion [25]. Furthermore, the introduction of deep learning techniques in steganography has greatly improved the effectiveness of steganographic concealment systems, resulting in a significant increase in security risks [47].

To combat this, researchers have turned to steganalysis, the process by which steganographic content is detected. In theory, steganalysis can be used in a security application that functions like a spam-filter to block malicious steganographic content from being transmitted across networks [18]. However, even though newer steganal-

Component	Usage
Steganographic Embedder	Hides data
Steganographic Decoder	Decodes hidden data
Steganalyzer	Detects hidden data

Table 1.1: List of definitions of various components of the steganography and steganalysis ecosystem.

ysis methods have shown promising results, they remain both relatively non-robust and impractical [6, 18]. In this thesis, we focus constructing a robust design and evaluation framework for practical and universal steganalysis.

1.1 Steganography and Steganalysis Ecosystem

In this section, we briefly introduce the parts listed in Table 1.1 that compose the steganography and steganalysis ecosystem.

1.1.1 Steganography

Figure 1-1: In a steganographic system, a secret message, M, is embedded into a cover image X via a steganographic embedder to generate a steganographic image, X_M. A steganographic decoder then decodes X_M to retrieve the message, M.

As described earlier, steganography is the process of hiding data in common file types. In this thesis, we are specifically concerned with image file types. Figure 1-1 shows a basic steganographic system, which uses the following process flow:

1. **Embedding.** A steganographic embedder is used to embed a secret message, M, into a cover image, X, to produce a steganographic image, X_M.

2. **Transmission.** The steganographic image X_M is transmitted to a steganographic decoder.

3. **Decoding.** The steganographic decoder decodes the steganographic image, X_M, and extracts the secret message, M.

In steganography, the steganographic embedder aims to minimize the difference between X and X_M. Steganographic embedders can be categorized as one of three types: (1) frequency [38], (2) spatial [15], and (3) deep learning [47]. The amount of data a steganographic embedder can transmit is known as the embedding ratio, which is the ratio between the sizes of M and X. Each steganographic embedder has a corresponding steganographic decoder that can decode the steganographic image generated by the steganographic embedder.

1.1.2 Steganalysis

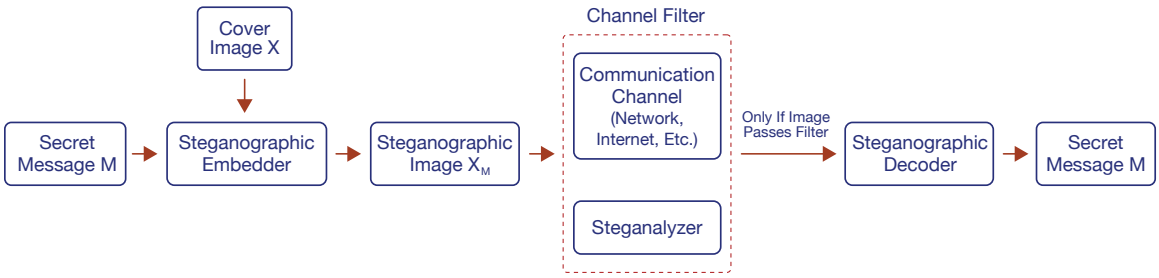

Figure 1-2: A steganographic embedder sends a message, M, by embedding it into a cover image, X, to produce X_M, and then transmits this image to an steganographic decoder. Steganalysis combats steganography by preventing images with steganographic content, X_M, from being transmitted. When steganalysis fails to filter out these images, the decoding process continues as normal and the steganographic decoder decodes the secret message, M.

Steganalysis is the process of detecting if a file contains steganographic content. Figure 1-2 shows how steganalyzers could be used in real-world scenarios to block the transmission of steganographic content. Steganalyzers often rely on either (1) statistical techniques [37] or (2) deep learning methods [6]. In recent years, deep learning steganalyzers have shown the most promising results and have significantly outperformed statistical steganalyzers [6].

Finally, steganalyzers are also classified as either (1) discriminate or (2) universal [6]. Discriminate steganalyzers can only detect a subset of steganographic embedders. On the other hand, universal steganalyzers are successful at detecting all known and unknown steganographic embedders. The eventual goal of steganalysis research is to produce a universal steganalyzer [6]. Universal steganalyzers are the key towards robust steganalysis for security applications.

1.2 Problem Definition and Challenges

In Appendix A, we show several steganography-enabled cybersecurity threat models along with applications that demonstrate the significant security risk posed by steganography. These threat models underscore the serious need for effective steganalyzers. As security risks associated with steganography continue to increase, researchers must focus on designing universal and practical steganalyzers that can be robustly deployed in defense networks. Without these characteristics, steganalyzers cannot effectively mitigate steganography-enabled threat models.

In this work, we only study deep learning based steganalysis since deep learning methods have shown the most promising and accurate results [6, 18]. Thus, we focus on identifying and solving critical problems that stand in the way of practical and universal deep learning steganalysis. Specifically, we identify the following problems:

1. **Training and Execution Efficiency Problem** - Most steganalyzers are computationally expensive to train and execute [6]. Practical steganalyzers should be computationally efficient so they can be effectively deployed in applications.

2. **Image Size Mismatch Problem Detection** - Most steganalyzers can only detect images of a certain size. Furthermore, even those that can detect images of varying sizes tend to be less accurate [10]. Practical steganalyzers must be effective at detecting images of all sizes.

3. **Source Mismatch Problem** - Steganalyzers commonly fail to detect steganographic images generated from a dataset the model has not been trained on [49].

Steganalyzers should be able to detect steganographic images regardless of the source dataset.

4. **Steganographic Embedder Mismatch Problem** - Most steganalyzers are discriminate steganalyzers and can only detect steganographic embedders they are trained on [40]. Steganalyzers must be able to detect unseen steganographic embedders to effectively mitigate steganography-enabled threat models.

5. **Low Embedding Ratio Problem** - Steganographic content is difficult to detect at low embedding ratios, making universal steganalysis increasingly challenging [9]. To neutralize steganography-enabled threat models, steganalyzers must be able to detect low embedding ratios effectively.

6. **Practical and Robust Evaluation Problem** - Current research only evaluates steganalysis in a limited context (i.e. steganalyzers are only tested on certain image datasets) [43]. Steganalyzers must be evaluated in a diverse context to ensure that they are robust and effective in real-world situations.

Each of these challenges must be solved in order to create practical and universal deep learning steganalyzers that can be effectively deployed in cybersecurity applications.

1.3 Contributions

In this work, we aim to solve several challenges facing practical and universal deep learning steganalysis. To this end, we explore several different robustness methodologies and draw insights from a comprehensive evaluation of state-of-the-art deep learning steganalyzers. Specifically, we make the following contributions:

- **Practical Steganalysis Architectures** - We design two architectures, ArbNet and FastNet, which enable image size mismatch problem detection and training and execution efficiency, respectively.

- **Robust Training Methodologies** - We propose robust training methods that help create more universal steganalyzers. These methods include data augmentation techniques that also help overcome our novel adversarial attack system, StegAttack.
- **Robust Steganalysis Evaluation System** - We develop a Python library, StegBench, that enables comprehensive evaluation of steganalysis.
- **Comprehensive Evaluation of Deep Learning Steganalyzers** - We comprehensively evaluate state-of-the-art deep learning steganalyzers to discover common failure modes and identify effective techniques.

1.4 Thesis Roadmap

The remainder of this thesis is organized as follows:

- Chapter 2 reviews key concepts and algorithms relevant for developing and evaluating steganography and steganalysis procedures.
- Chapter 3 describes robust methodologies for universal and practical steganalysis and proposes new deep learning steganalysis architectures.
- Chapter 4 covers the design of our steganalysis evaluation system, StegBench.
- Chapter 5 details and discusses extensive results generated by our experiments.
- Chapter 6 describes our novel adversarial attack system, StegAttack.
- Chapter 7 provides concluding remarks and future directions.
- Appendix A discusses our design of malware-based steganographic threat models and describes a suite of systems developed to demo these threat models.
- Appendix B contains raw data from our experiments.
- Appendix C gives details on StegBench configurations and the StegBench API.

Chapter 2

Background and Related Work

In this chapter, we provide the requisite background for the rest of the thesis. First, we introduce deep learning methods and several deep learning architectures. Then, we explain the fundamentals of steganography and steganalysis. Finally, we review steganalysis research related to the work presented in this thesis.

2.1 Deep Learning Overview

Broadly, deep learning is defined as a class of machine learning algorithms that use multi-layered neural networks to extract higher-level features from raw data [11]. These methods use large amounts of training data to extract complicated, feature-rich data for either generative or classification tasks [11]. In this thesis, we use a number of deep learning approaches to improve our steganalyzers.

2.1.1 Generative Adversarial Network

A generative adversarial network (GAN) is a deep learning architecture comprised of two neural networks: a generator and a discriminator [12]. The generator learns to generate plausible data, which act as negative training examples for the discriminator. When training begins, the generator produces obviously fake data, and the discriminator quickly learns to tell that it is fake, penalizing the generator for produc-

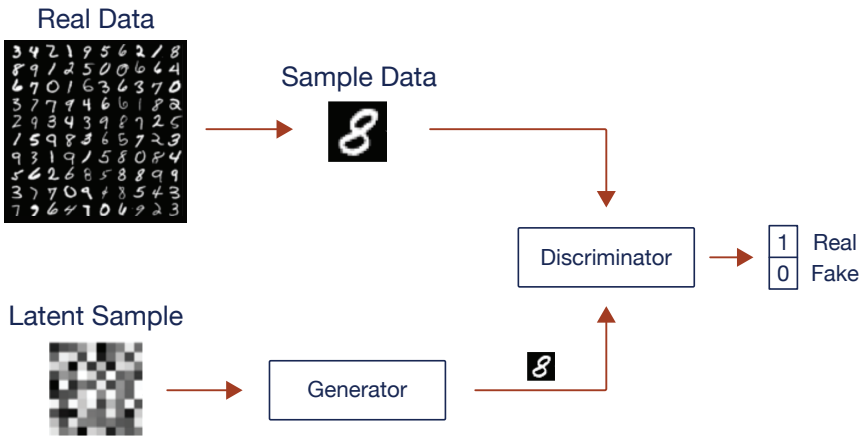

Figure 2-1: Generative adversarial networks (GANs) are deep learning architectures that are composed of a generator and a discriminator. The generator learns how to transform random noise into a target distribution, while the discriminator attempts to identify if the generated images are fake or real. This adversarial setup allows the generator to effectively model target distributions.

ing implausible results. Over time, the generator gets better at generating realistic-looking data while the discriminator learns to better distinguish the generator's fake data from real data.

Figure 2-1 shows an example GAN system in which the generator is learning to produce fake handwritten Arabic numerals and the discriminator is tasked with determining whether the images were produced by a human or the generator. GAN systems are increasingly used in steganography since they are excellent at generating hard-to-detect steganographic images [47].

2.1.2 Convolutional Neural Network

A convolutional neural network (CNN) is a deep learning architecture capable of taking an input image, assigning importance (learnable weights and biases) to various aspects/objects in the image, and then using these signals for image classification. Figure 2-2 shows an example CNN, which is composed of several specialized layers. CNNs are used extensively in steganalysis for their ability to learn important features relevant to detecting steganographic images [6]. Since we use CNNs extensively in

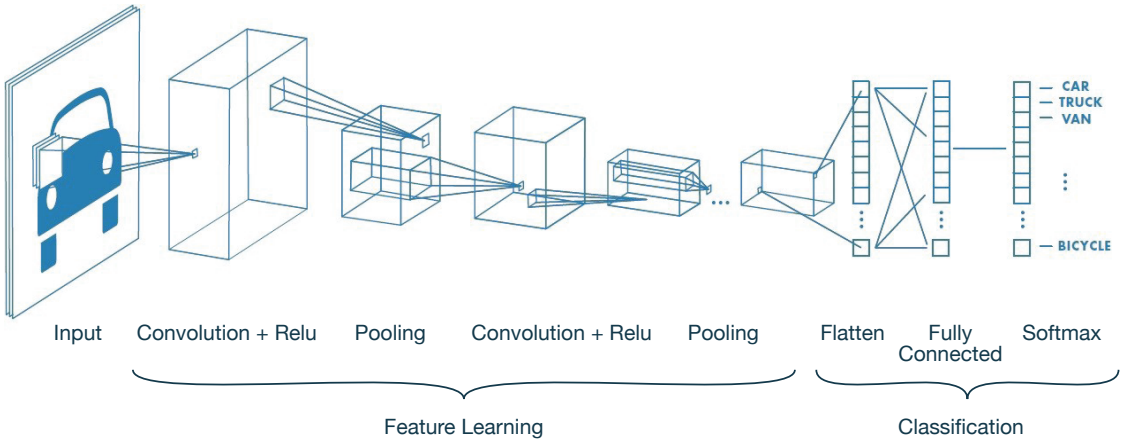

Figure 2-2: Convolutional neural networks (CNNs) are deep learning architectures that use convolutional layers and sub-sampling layers to extract and classify meaningful signals from image data. Image Credit: MathWorks[1]

this thesis, we now describe several of the specialized layers shown in Figure 2-2.

Convolution Layer

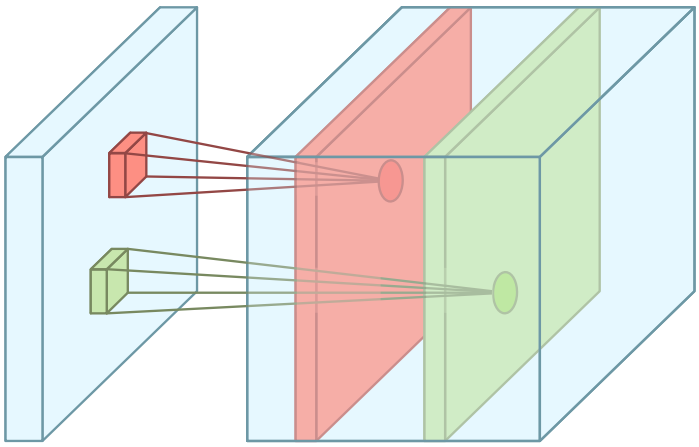

Figure 2-3: Convolutional layers extract spatially relevant data by convolving weights with input data. Convolutions are defined by their kernel size, stride, and padding. These parameters affect the types of spatial features the convolution can extract.

The convolutional layer is the core building block of a CNN. The layer's parameters consist of a set of learnable filters (or kernels), each of which has a small receptive field but extends through the full depth of the input volume. Figure 2-3 shows an example convolution, showcasing how convolutions extract spatially-relevant data.

[1] `https://www.mathworks.com/`

Pooling Layer

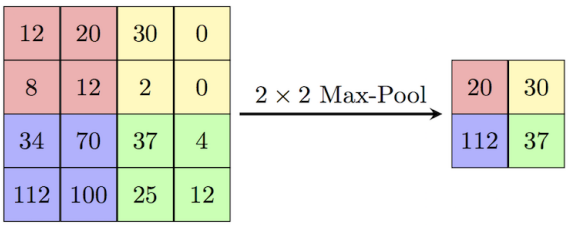

Figure 2-4: Pooling layers down-sample input data by applying a function on sub-regions. For example, max pooling applies a max function to extract the strongest sub-region signals. Pooling is effective at reducing noise and extracting larger features. Image Credit: CS Wiki[2]

The pooling layer is a form of non-linear down-sampling. There are several non-linear functions for implementing pooling, among which max pooling is the most common. It partitions the input image into a set of non-overlapping rectangles and outputs the maximum for each such sub-region. Figure 2-4 shows how max pooling extracts the maximal signal from its receptive field.

Activation Unit

$$f(x) = \max(0, x) \tag{2.1}$$

The activation unit aids in the non-linear decision making of the system by allowing certain inputs to be sent forward. Traditionally, this unit is a rectified linear unit (ReLU), which applies the non-saturating activation function shown in Eq. 2.1

$$\texttt{TLU:} \begin{cases} -T & x < -T \\ x & -T \leq x \leq T \\ T & x > T \end{cases} \tag{2.2}$$

Even though ReLU is a popular choice for image classification, we also experiment with the truncated linear unit (TLU) [44], shown in Eq. 2.2, where T is heuristically chosen. TLU's are much more effective at boosting weak signals, and thus well suited

[2] `https://computersciencewiki.org/`

for a weak signal-to-noise ratio environment like steganalysis [44].

2.1.3 Model Evaluation

In this section, we review key details related to the evaluation of deep learning models.

Data

When evaluating models, it is important that data is generated or collected carefully. Since a deep learning model's success is directly related to the composition of the data on which they are trained and tested, it is crucial that models are evaluated in the context of their training/test data. For example, it is important to compare different models across the same training/test data to enable a fair comparison. Furthermore, careful consideration should be taken to create a diverse test set so that confounding variables such as source distribution do not affect model performance. In summary, models should always be evaluated in the context of the data that they were trained and tested on.

Metrics

In our experiments, to measure steganalyzer performance, we report the blind detection error at an unoptimized threshold of 0.5. To calculate this error, we first classify all the results using the threshold and then use the following equation:

$$ERROR = \frac{FP + FN}{FP + TP + FN + TN} \tag{2.3}$$

Eq. 2.3 shows the calculation for the total detection error, where FP is the number of false positives, FN is the number of false negatives, TP is the number of true positives, and TN is the number of true negatives. The detection error represents the percentage of incorrect classifications made by the model.

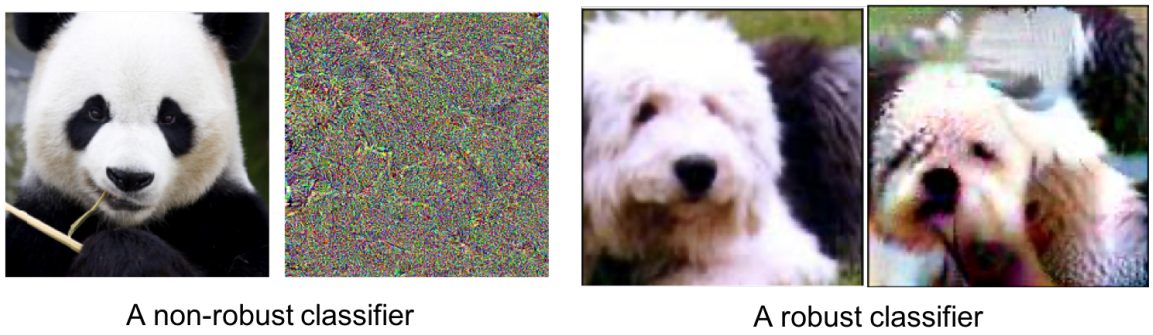

Figure 2-5: The left-hand side shows a non-robust classifier that uses noisy signal to classify an image. The right-hand side of the figure shows how a robust classifier uses human-meaningful features and signals to classify an image. Even though non-robust classifiers achieve high accuracy, their use of non-robust signal leaves them open to adversarial attacks. In comparison, robust classifiers use meaningful signal and are reliable. Image Adapted from: Ref. [8]

Model Robustness

Robustness is akin to the mathematical concept of stability, which is defined as how effective a model is when tested on a slightly perturbed version of a clean input, where the outcome is supposed to be the same [32]. As found by Athalye et al., many machine learning models are relatively non-robust. Specifically, these models pick up useful but non-robust signals that translate to noisy features [1]. Figure 2-5 shows how a robust classifier uses relevant features while a non-robust classifier uses non-robust noise as its signal.

Because of this, it is possible to add small adversarial perturbations to input data to generate adversarial images which the model misclassifies [1]. This technique of adding adversarial perturbations to input data is known as an adversarial attack. Figure 2-6 shows an example adversarial attack that uses the fast gradient sign method to generate an adversarial image.

In this thesis, we make extensive use of adversarial attacks to generate adversarial images for steganalyzers. Current research shows that adversarial images are useful as training samples and help create robust models [51]. Researchers argue that training on adversarial images allow deep learning models to learn which signals are robust and which are not [51]. Below, we outline several adversarial attack methods that

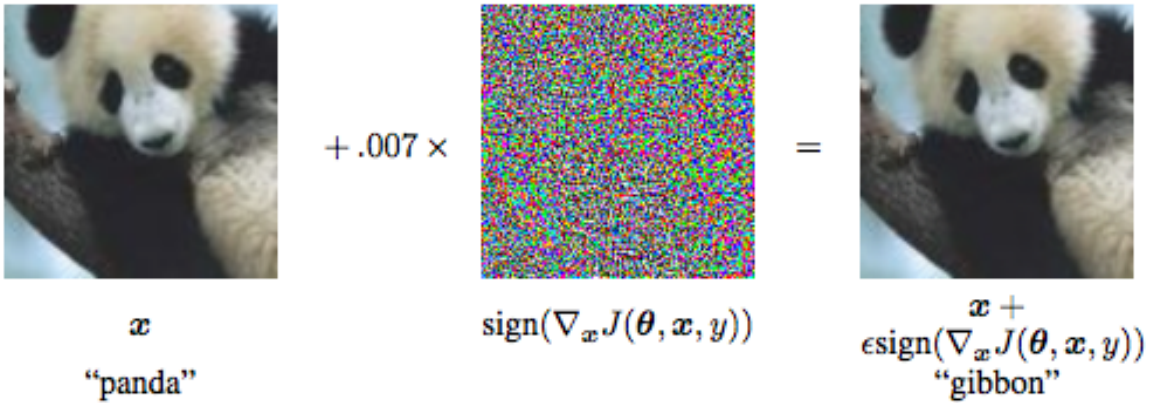

Figure 2-6: Non-robust classifiers can be attacked using tactics such as the fast gradient sign method (FGSM). FGSM adds noise using the sign of the loss gradient of the model with respect to the input image to create an adversarial example. Non-robust models tend to misclassify these adversarial examples (i.e. a panda as a gibbon), even though the image is relatively unchanged. Image Credit: OpenAI[3]

used in this thesis.

$$X^{adv} = X + \epsilon \cdot sign(\nabla_\theta J(X, \theta)) \tag{2.4}$$

Eq. 2.4 shows the fast gradient sign method (FGSM) [13], where X is the original image, X^{adv} is the adversarial image, ϵ is the step-size, θ is the model, and $sign(\nabla_\theta J(X, \theta))$ is the sign of the deep learning model's gradient along the input image. FGSM uses a single-step to produce the adversarial image.

$$X^{adv}_{N+1} = X^{adv}_N - \epsilon \cdot \Pi_X(X^{adv}_N - \nabla_\theta J(X^{adv}_N, \theta)) \tag{2.5}$$

Eq. 2.5 shows the projected gradient descent method (PGD) [24], where X^{adv}_i is the ith iteration of the adversarial image, ϵ is the step-size, θ is the model, $\nabla_\theta J(X^{adv}_N, \theta)$ is the deep learning model's gradient along the Nth adversarial image, and $\Pi_X(...)$ is a function that projects onto the feasible set, X, which is usually a constrained l_p space. PGD is an iterative process that updates the adversarial image with the projected gradient.

Each attack method is used to generate an adversarial perturbation that causes

[3] `https://openai.com/`

the model to incorrectly classify the adversarial input. PGD is much more effective compared to FGSM since it iteratively uses the model's gradient to find the most effective adversarial perturbation.

2.2 Steganography Techniques

Steganography is the procedure of concealing data inside other file types. For the purposes of this thesis, we focus exclusively on steganography in the image domain. Most steganographic embedders can be categorized as (1) frequency, (2) spatial, or (3) deep learning [26, 47]. Frequency domain steganographic embedders use statistical techniques to hide information in the frequency coefficients of an image [20]. Spatial domain steganographic embedders also use statistical techniques but hide information in the raw pixel bits of an image [26]. In comparison, deep learning steganographic embedders use deep learning architectures such as GANs to hide information in an image [2, 47]. Deep learning steganographic embedders have been very effective at hiding large quantities of data while avoiding detection from state-of-the-art steganalyzers [47]. Finally, it is important to note that every steganographic embedder has a corresponding steganographic decoder that decodes the embedded message from the steganographic image.

Steganography is measured using an embedding ratio. This measurement specifies the ratio between the embedded data size and the cover image size [20]. The units of this measurement are specific to each type of steganography. We define the embedding ratio units for each steganographic embedder type in the following sections.

2.2.1 Frequency Domain

The frequency domain refers to images represented by signal data, such as the JPEG image format. In this domain, raw image data (pixel values) are translated to signal data via some signal processing method (i.e. discrete cosine transform) to produce a set of signal coefficients (i.e. discrete cosine transform coefficients).

Figure 2-7 shows how pixel values are translated into DCT coefficients. While spa-

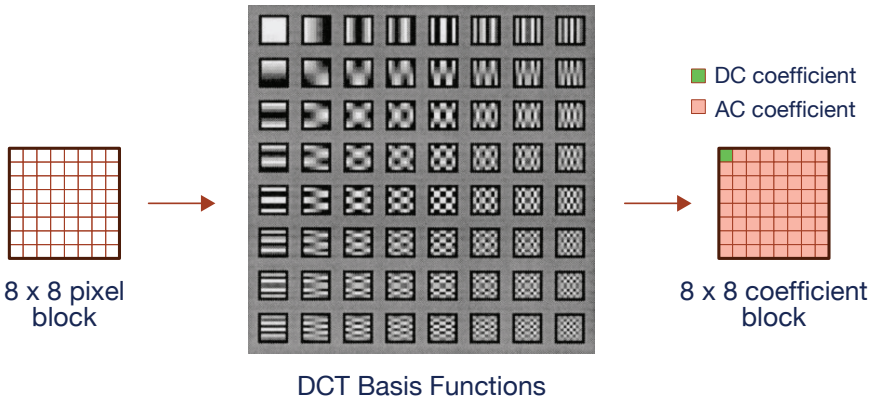

Figure 2-7: In the frequency domain, image data is stored in quantized frequency coefficients. Frequency-domain steganography embeds message data by modifying the non-zero frequency coefficients of the image. Image Credit: EE Times[4]

tial domain steganography operates on pixel values (the left array), frequency domain steganography operates on DCT coefficients (the right array). Common methods in frequency-based steganography include F5 [38], J_UNIWARD [16], EBS [36], and UED [14]. For the most part, these methods aim to minimize statistical distortions in the steganographic image.

The embedding ratio of frequency domain steganography is measured in bits per non-zero AC DCT coefficient (bpnzAC) [20]. The AC coefficients represent 63 total coefficients in each coefficient block, excluding the coefficient at [0, 0] (i.e. 239 in Fig. 2-7). Traditionally, the coefficient at [0, 0] holds the most signal in that coefficient block and is never modified. Furthermore, zero-valued coefficients are also not counted, since most steganographic embedders avoid using these coefficients.

2.2.2 Spatial Domain

The spatial domain is defined as the raw image pixels that are used to define an image. For png images, this is the RGB value used to represent each pixel. Spatial domain steganography conceals the secret information within these pixel values, usually by substituting secret bits inside them [20].

Figure 2-8 shows an example of the least-significant bit (LSB) method, which

[4] https://www.eetimes.com/baseline-jpeg-compression-juggles-image-quality-and-size/

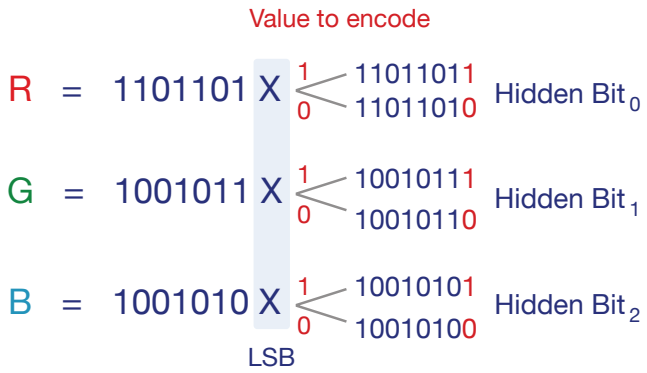

Figure 2-8: Least significant bit (LSB) steganography is a spatial technique that replaces the last bit of the pixel with message content. For example, if the value being encoded is a 1, the last pixel value bit is set to 1. Image Credit: KitPloit[5]

embeds message content into the final bit of each of the RGB channels [20]. Other methods include S_UNIWARD [16], HUGO [29], WOW [15], and HILL [21]. These methods use sophisticated techniques that minimize image distortion to better embed data, thereby reducing the impact that the embedding operation has on the underlying source distribution of the cover image. The embedding ratio of spatial domain steganography is measured in bits per pixel (bpp) [20].

2.2.3 Deep Learning Domain

The deep learning domain refers to steganographic embedders that make use of deep learning methods to embed steganographic content. Even though deep learning techniques operate on either the spatial or frequency domain, we have intentionally separated them into their own category because they are functionally very different. Traditionally, steganographic embedders use generative deep learning networks to combine an input message and cover image into a steganographic image [47]. Still, there are many variations on network designs that have been proposed such as SteganoGAN [47], HiDDeN [52], and BNet [2]. Figure 2-9 shows SteganoGAN [47], which makes use of a GAN architecture to generate high-quality steganographic images.

The embedding ratio for deep learning steganography is specific to the architecture that is being used. When reporting embedding ratios, researchers must be careful

[5] `http://kitploit.com/`

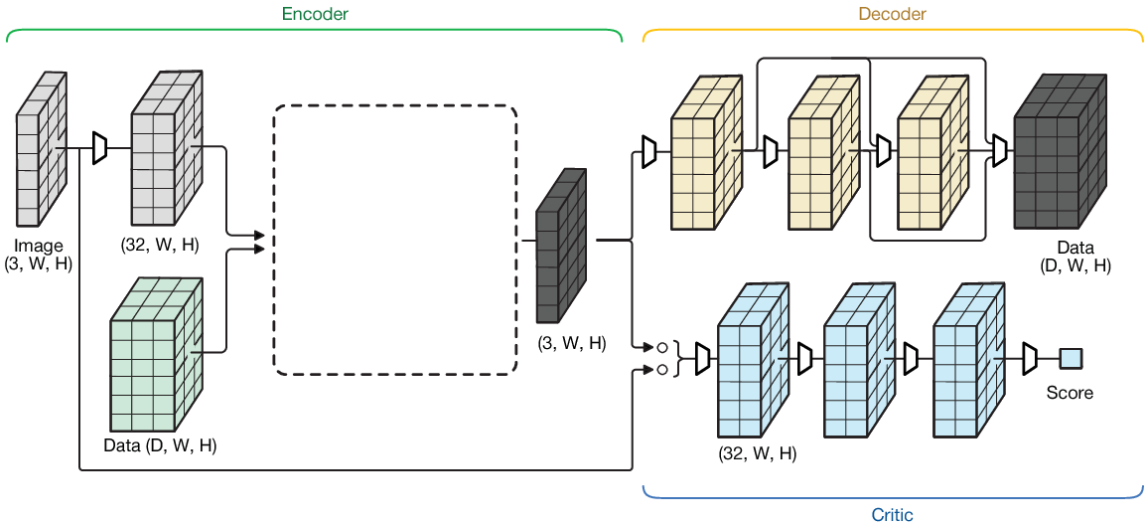

Figure 2-9: Machine learning-based steganography uses machine learning to embed data inside images. This figure shows SteganoGAN, which is a generative adversarial network that embeds messages in a highly undetectable way. Image Credit: Ref. [47]

to make sure that the deep learning embedding ratio is equivalent to the traditional measures found in the spatial and frequency domain. For the purposes of this thesis, we use Reed Solomon [39] bits per pixel, a measure introduced by [47], to measure the embedding ratio of deep learning steganographic embedders. Reed Solomon bpp measures how much real data is transmitted by the steganographic image by calculating the probability of a bit being recovered correctly by the deep learning steganographic decoder [47]. This measure allows for equivalent comparison to the bpp measurement found in the spatial domain.

2.3 Steganalysis Techniques

Steganalysis is the process that detects whether a file contains steganographic content. Steganalysis procedures fall into one of two categories: (1) statistical steganalyzers or (2) deep learning steganalyzers [6]. In general, statistical steganalyzers use extensive feature engineering to exploit the fact that steganographic embedders introduce statistically significant artifacts (i.e. pixel bit values follow unnatural distributions) [37]. In comparison, deep learning steganalyzers use CNN architectures to learn the statistical imprints of different steganographic embedders [4, 44]. These methods have

proven to be very effective against spatial and frequency modes of steganography [6].

2.3.1 Statistical Techniques

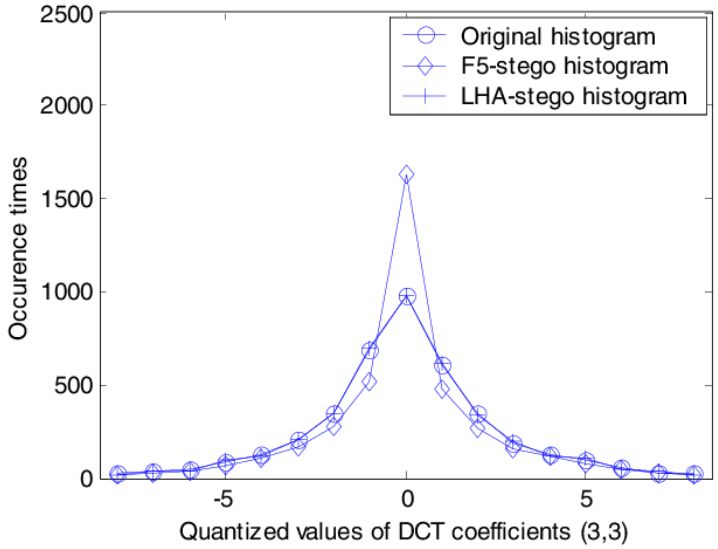

Figure 2-10: steganographic embedders leave statistical artifacts in the steganographic image. In this figure, the F5 algorithm leaves a noticeable effect on the quantized DCT histogram. Statistical steganalyzers use statistical signals such as statistical moments to infer if an image is steganographic. Image credit: Ref. [48]

Statistical steganalyzers rely on detecting statistical abnormalities introduced via steganographic embedders. For example, LSB steganography has been shown to significantly modify an image's natural pixel value distribution [6, 20]. Figure 2-10 shows how, even for small embedding ratios in a JPEG image, steganographic embedders such as F5 [38] introduce slight changes in the distribution of underlying coefficient values [48].

Statistical steganalyzers are often augmented with support-vector machines and specialized feature-engineering to pull out rich signal data from the underlying source distribution [37]. In general, statistical steganalyzers are only effective against a limited set of spatial and frequency-based steganographic embedders [23].

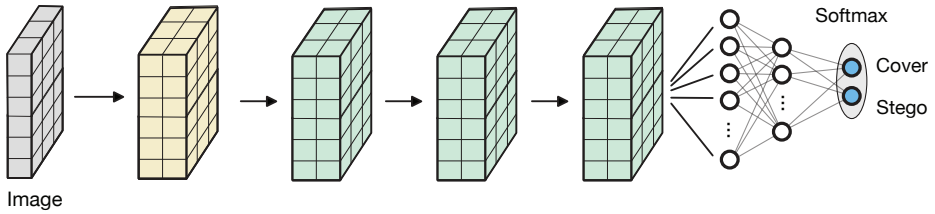

Figure 2-11: Deep learning based steganalyzers use deep learning convolutional neural networks to extract useful signal from the images to determine if it is steganographic.

2.3.2 Deep Learning Techniques

Deep learning steganalyzers traditionally use CNN architectures to extract steganographic signal [6]. Figure 2-11 shows a simple CNN steganalyzer. In recent years, many deep learning architectures have been proposed for steganalysis such as YeNet [44], XuNet [42], SRNet [5], and CISNet [41].

Deep learning steganalyzers depend heavily on their training dataset [49]. They have been shown to be the most effective tools for steganalysis when compared to statistical steganalyzers [6]. Yet, even though deep learning steganalyzers have been very effective at steganographic detection, they remain relatively non-robust. In fact, recent papers have shown that deep learning steganalyzers have a large number of failure modes [50]. Understanding and solving these failure modes remains one of the more challenging obstacles facing deep learning steganalyzers.

2.4 Related Work

In recent years, the development of deep convolutional neural networks has pushed the boundaries of steganalysis research [6]. CNN architectures provide steganalyzers with a larger feature space to extract more useful signals from image data [40]. In 2015, Qian et al. published a simple deep learning model that showcased the potential for CNN-based steganalysis [31]. Then, in 2016, Xu et al. was able to achieve state-of-the art results using a more complicated network structure [42]. In 2018, researchers released several new architectures, including Ye-Net [44], which used truncated linear activation functions, Yedroudj-Net [45], which used a small network that could learn

with small datasets to reduce computational costs, and SRNet [5], which could be adapted to spatial or frequency steganalysis.

To date, most research in the steganalysis domain has focused on (1) developing architectures that boost steganographic signal, since steganalysis operates in a low signal to noise ratio environment [40]; and (2) developing architectures with better convergence guarantees and reduced computational costs [6]. Yet even though architectures have become progressively more sophisticated, model robustness has not seen comparable improvements. Recently, researchers have found that many steganalyzers previously considered to be state-of-the-art are particularly non-robust and have a number of failure modes [46, 50]. Specifically, these steganalyzers fail at the source mismatch problem, low embedding ratio problem, and steganographic embedder mismatch problem. They are also prone to adversarial attacks.

Next, researchers have also worked on understanding how to robustly evaluate steganalyzers. Recent work has taken two approaches to tackling the evaluation issue: (1) formulating standardized methodologies that lead to robust evaluation, and (2) the design and generation of steganographic datasets that allow for evaluation.

In the first approach, researchers have done comprehensive reviews of the current steganographic evaluation ecosystem, summarized flaws regarding current evaluation schemes, and proposed alternative methods for more robust evaluation [6, 23, 30, 46]. Zeng et al. finds that current steganalyzers are particularly bad at the source mismatch problem and argues that steganalyzers should be evaluated on increasingly diverse source distributions [46]. Prokhozhev et al. proposes several tactics to improve evaluation methods involving steganographic embedder mismatch problem and the use of large test datasets [30]. Finally, Chaumont et al. finds that researchers should test steganalyzers on more diverse datasets (i.e. image source, image resolution, image format) to evaluate how practical a steganalyzer is [6].

In the second approach, researchers focus on the design and generation of steganographic datasets. When designing these datasets, researchers focus on (1) cover image sources and (2) steganographic embedder diversity. The most commonly used evaluation dataset is the BOSS dataset [3], which contains gray-scale 512x512 images. More

recent work has criticized widespread use of BOSS since it contains limited image diversity [6]. Researchers have continued to develop a number of new datasets, including SIG [35], StegoAppDB [27], and iStego100k [43]. Yet, because these datasets are unchanged and static, their usability diminishes over time and they are rarely adopted by the steganalysis research community.

In this thesis, we build off the related work discussed here and focus on designing a robust framework for practical and universal steganalysis that includes practical design considerations, robust training methodologies and a dynamic, effective evaluation system. Our work builds on the insights of previous research and brings many of their suggested ideas and methodologies into practice.

Chapter 3

Universal and Practical Steganalysis

In this chapter, we focus on designing methodologies that help create universal and practical steganalyzers. In section 3.1, we list the problems facing universal steganalysis. In section 3.2, we list the problems facing practical steganalysis. In section 3.3, we specify several data augmentation techniques that help train universal steganalyzers. Finally, in section 3.4, we design two new deep learning architectures: ArbNet and FastNet. ArbNet provides a solution to the image size mismatch problem while FastNet provides a solution to the training and execution efficiency problem.

3.1 Towards Universal Steganalysis

Universal steganalysis is defined as a class of steganalyzers that can detect any known or unknown steganographic embedder [6]. We identify the following problems that a universal steganalysis solution must solve:

- **Source Mismatch Problem** - Universal steganalyzers must be able to detect steganographic images regardless of the image source.
- **Steganographic Embedder Mismatch Problem** - Universal steganalyzers must be able to detect steganographic images from any steganographic embed-

der, including those that they have not been seen before.

- **Low Embedding Ratio Problem** - Universal steganalyzers must be able to detect steganographic images that are embedded with low embedding ratios.

By solving these problems, a steganalyzer would be able to function as a universal steganalyzer in any context. Next, we find that from the perspective of deep learning, universal steganalysis is related to model robustness. Specifically, if we assume that training on a constrained set of steganographic signals is sufficient for learning any steganographic signal, then a deep learning steganalyzer's robustness is related to how effectively a model can detect unseen steganographic images based off a limited training dataset. Thus, to build universal deep learning-based steganalyzers, we should pay close attention to training datasets, as they could be used to effectively create universal steganalyzers.

3.2 Towards Practical Steganalysis

Practical steganalyzers must be effective in real-world situations. From a deep learning perspective, this means that models must be efficient and applicable in a diverse set of contexts. Towards this goal, we identify the following problems that a practical steganalysis solution must solve:

- **Image Size Mismatch Problem** - Practical steganalyzers must be able to operate on arbitrary image sizes (i.e. 256x256, 512x512, etc.) so that they can be usefully deployed in an application.

- **Training and Execution Efficiency Problem** - Practical steganalyzers must be able to be trained and updated efficiently so that new training examples can be quickly incorporated into the model. They must also be able to execute efficiently so that they can process a large of number of images quickly.

3.3 Dataset Augmentation

In this section, we describe data augmentation procedures that generate datasets that can be used to train universal deep learning steganalyzers. Data augmentation is a technique used to artificially expand the training dataset by introducing image modifications that increase the diversity of the dataset. Research in the last few years has shown that data augmentation effectively improves the robustness of deep learning models by providing useful training features [7, 51].

Through careful literature review and our own experiments, which we detail in Chapter 5, we design the following data augmentation procedures that researchers should consider using to augment their steganographic training datasets: (1) source diversity, (2) steganographic embedder diversity, and (3) embedding ratio diversity.

3.3.1 Source Diversity

To solve for the source mismatch problem, we suggest augmenting training datasets with a large sampling of source distributions. We suggest the following data augmentations:

- A large variety of camera configurations should be present in the dataset to ensure that that the training dataset contains many source distributions.
- A large variety of image sizes and image formats should be present in the dataset to ensure that the training dataset contains a diverse set of source image types.
- Standard data augmentation techniques such as random crops, rotations, and translations should be employed to artificially increase source diversity.

By using a diverse set of training images, source distribution signals that confound steganographic signals will be less useful for prediction. Consequently, steganalyzers trained on these datasets will better learn to use actual steganographic signals which will allow them to effectively detect steganographic images from other sources, thereby helping solve the source mismatch problem.

3.3.2 Steganographic Embedder Diversity

To solve for the steganographic embedder mismatch problem, we suggest augmenting training datasets with a diverse set of steganographic embedders. We suggest the following data augmentations:

- Datasets should skew towards using hard-to-detect steganographic embedders (i.e. SteganoGAN [47]), since they create a more challenging steganalysis problem and provide more useful steganographic signal.

- To test universal steganalyzers, the test dataset should be augmented with steganographic embedders that have not been trained on and, preferably, are characteristically different from the training dataset steganographic embedders.

Following these guidelines will enable deep learning-based steganalysis architectures to effectively learn to detect steganographic signals commonly found among various steganographic embedders. As steganography improves, the suggested steganographic embedder composition of these training datasets will also evolve.

3.3.3 Embedding Ratio Diversity

To solve for the low embedding ratio problem, we suggest augmenting training datasets with a diverse set of embedding ratios. We suggest the following data augmentation:

- A large range of embedding ratios should be used for training so that models do not over-fit to a specific embedding ratio. We suggest a range from 0.05 bpp to 1.0 bpp. For deep learning steganographic embedders, researchers should consider increasing the range of embedding ratios present in the training dataset.

Using a wide range of embedding ratios will force models to better learn to detect steganographic signal and prevent overfitting for a specific embedding ratio. Having the aid of steganographic images with larger embedding ratios during training will enable steganalyzers to identify and boost steganographic signal present in steganographic images with lower embedding ratios.

3.4 Architectures

In this section, we describe two deep learning architectures that provide potential solutions to the challenges facing practical steganalysis. In section 3.2, we identified that a practical deep learning steganalyzer must be able to solve the following problems: the image size mismatch problem and the training and execution efficiency problem.

To help realize these goals, we use modern CNN methods to design: (1) ArbNet, which is compatible with arbitrary input image sizes using a modified DenseNet architecture, and (2) FastNet, which uses a modified EfficientNet architecture to improve computational efficiency.

3.4.1 ArbNet

To solve for the image size mismatch problem, we introduce ArbNet (Arbitrary Image Network), which uses several modifications to the architecture presented in Singh et al. [33]. Singh et al. presents a DenseNet [17] structure adapted to fit the steganalysis problem. We use this architecture and modify it in the following ways:

1. **TLU Activation Units.** TLU activation units (section 2.12) are used instead of ReLU to extract weak steganographic signal. The threshold is set to three as per [44].

2. **Modified ConvBlocks.** The number of convolutional blocks and specific convolutional layer parameters are modified as shown in Figure 3-2.

3. **Spatial Pyramid Pooling**. Spatial pyramid pooling (SPP), shown in Figure 3-1, is used instead of global average pooling to boost steganographic signal and enable arbitrary image size detection. SPP produces fixed-length representations at various scales that are invariant to input size.

Figure 3-2 shows the overall architecture of ArbNet. In the first part of the network, we preprocess the input image with a set of learnable filters that are initialized

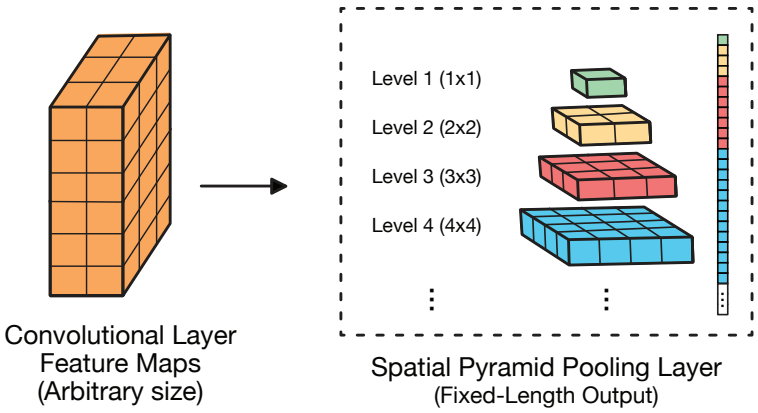

Figure 3-1: A spatial pyramid pooling (SPP) layer is invariant to input size since it fixes the output size of the pooling layer. Each SPP layer fixes how many sections the input will be divided into for pooling. In this figure, the third layer divides the convolution input into nine pooling sections

as SRM high pass filters. We then combine this output with the raw input data and feed it into the DenseNet structure. The DenseNet architecture extracts residual signal using skip connections. Next, the residual output is combined with the original image data and fed into the spatial pyramid pooling layer. Finally, the output from the spatial pyramid pooling layer is fed into the fully connected layer to produce the classification. Each step helps boost weak steganographic signal and makes the model effective in a low signal-to-noise ratio environment. We believe that ArbNet holds a key solution to the image scaling problem as it effectively avoids downsampling input signal while enabling arbitrary image size detection.

3.4.2 FastNet

To solve for the training and execution efficiency problem, we introduce FastNet (Fast Network), which adapts EfficientNets [34] to the steganalysis problem. EfficientNets are a family of image classification models which achieve state-of-the-art accuracy but are an order of magnitude smaller and faster than previous models [34]. EfficientNets are trained by successively improving older models through a model-scaling process that modifies network depth, width, and resolution [34]. EfficientNets use MBConv blocks, which can be used for neural architectural search, enabling EfficientNet to use the most effective neural architecture [34].

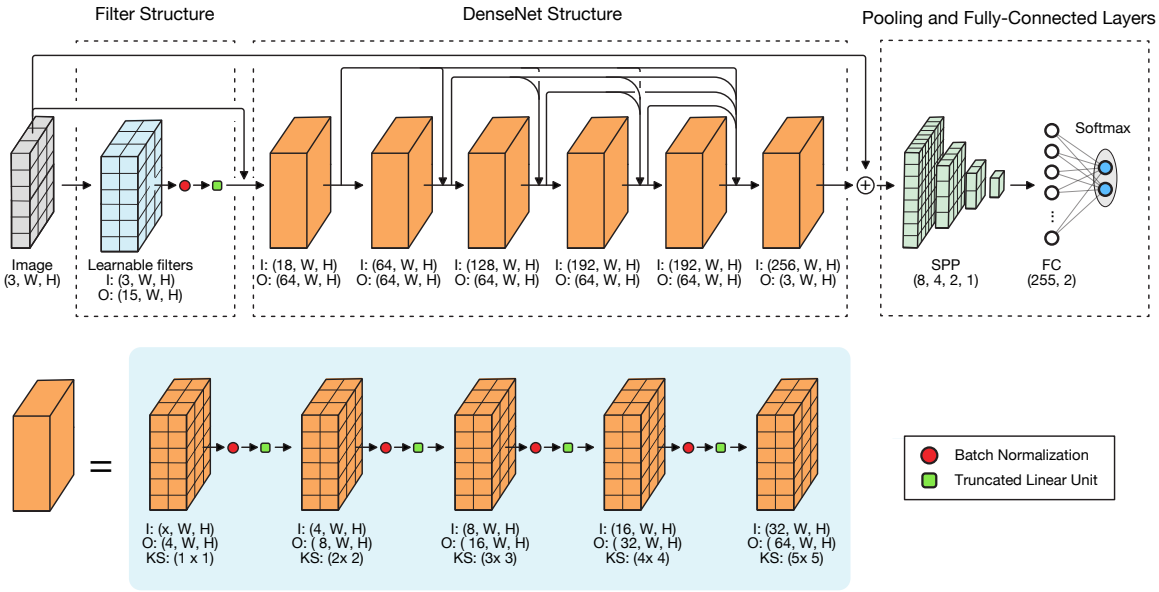

Figure 3-2: ArbNet is a steganalysis CNN model that can be applied to arbitrary image sizes. First, the input image is fed through 15 filters that are initially initialized with 15 SRM HPF's. Next, the input image and filtered output are fed into a DenseNet structure. Finally, the residual output from the DenseNet is combined with the original image and fed into a spatial pyramid pooling layer, a fully-connected layer, and a softmax function to output the steganographic and cover probabilities.

Figure 3-3 shows the overall architecture of FastNet. FastNet uses the EfficientNet-B0 network presented in [34], which is the baseline EfficientNet architecture for ImageNet classification. FastNet introduces slight modifications to the EfficientNet-B0 architecture to adapt the network to the steganalysis problem. Specifically, FastNet modifies the number of channels in the final MBConv block and the number of nodes in the fully connected layer.

Next, even though the best accuracy with EfficientNet is reported when the model's architecture is successively improved, we believe that using the original B0 network is a more effective means of showcasing the potential of EfficientNets. Specifically, if the adapted-B0 network, FastNet, is able to perform reasonably on the steganalysis challenge, we can be quite confident that EfficientNets hold a key solution to the training and execution efficiency problem.

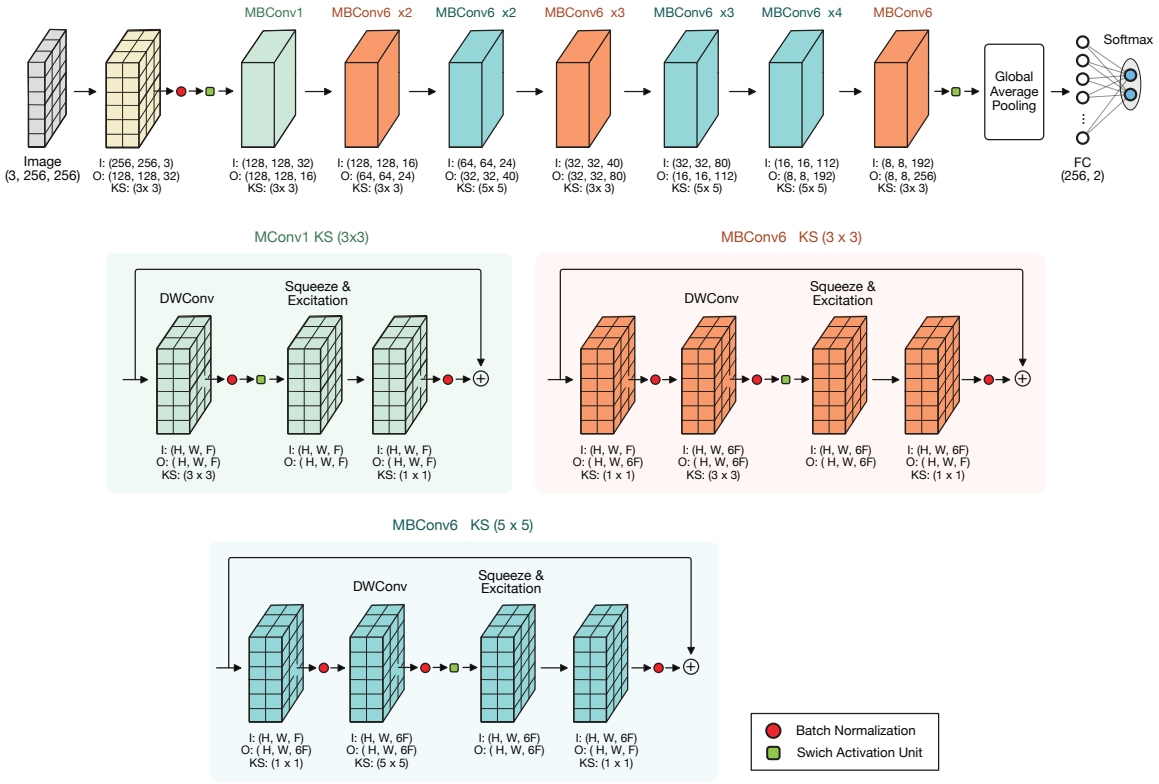

Figure 3-3: FastNet uses the EfficientNet-B0 structure and modifies the final few layers to adapt the network to the steganalysis problem. FastNet is composed of MBConv blocks which enable neural architecture search and computational efficiency.

Chapter 4

Benchmarking and Evaluation System for Steganalysis

In this chapter, we describe the development and usage of StegBench, a Python library that enables the robust evaluation of steganalysis. StegBench can seamlessly integrate into existing steganography and steganalysis evaluation setups through its powerful and efficient configuration management platform. StegBench is currently under active development and is available upon request.

In section 4.1, we outline the minimum system criteria that StegBench must satisfy. In section 4.2, we list design goals that guide our overall implementation. In section 4.3, we describe the architecture of StegBench along with extensive descriptions and usage patterns for each of the core modules. For more information on the StegBench system and API specifications, we refer the reader to Appendix C.

4.1 System Criteria

In this section, we detail specific system requirements that an effective steganalysis evaluation system must satisfy. We find that a steganalysis evaluation system should be able to effectively achieve the following: (1) steganographic dataset generation and (2) standard steganalysis evaluation. For each of these requirements, we further delineate a set of criteria that StegBench must satisfy. Together, these criteria are

sufficient for enabling effective steganalysis evaluation.

4.1.1 Steganographic Dataset Generation

Steganographic embedders and steganalyzers are evaluated by how well they either embed or detect data in a transmission medium. Thus, proper data processing and data generation is critical for effective evaluation since the transmission medium is key to the steganographic system.

Steganalysis evaluation datasets are defined by three key parameters: the source dataset, the steganographic embedder, and the embedding ratio. In our system, we would like to generate diverse steganographic datasets by modifying each of these parameters. Hence, we outline the following three criteria that our system must meet:

1. **Source Diversity:** The system must be able to access a large, diverse set of source distributions. In addition, the system must provide a large selection of dataset and image processing tools for data augmentation.

2. **Steganographic Embedder Diversity:** The system must integrate with a large and diverse set of steganographic embedders.

3. **Embedding Ratio Diversity:** The system must be able to apply steganographic embedders with a wide selection of embedding ratios.

4.1.2 Standard Steganalysis Evaluation

For a steganalysis evaluation system to be useful, it must adhere to standard evaluation methodologies for binary classification tasks and produce reproducible metrics that can be fairly compared to other steganalysis research. To achieve these requirements, we identify the following criteria:

1. **Comparable Metrics:** The system must produce standard binary classification metrics that are accurate and comparable to other steganalysis research to enable meaningful and fair comparisons.

2. **Reproducible Metrics:** The system must produce reproducible metrics so that steganalysis performance can be fairly reviewed.

3. **Steganalyzer Diversity:** The system must integrate with a large and diverse set of steganalyzers to enable comprehensive and robust comparisons.

4.2 Design Goals

In this section, we describe design goals for our steganalysis evaluation system, StegBench. With the aim of producing a usable, effective, and highly-compatible system, we adopt the following design goals:

- **Modularity:** StegBench must be properly subdivided into modules that are easily modified and interchanged.
- **Efficiency:** StegBench must use modern computational practices.
- **Compatibility:** StegBench must be compatible with a large number of steganographic embedders, steganalyzers, and execution environments.

4.3 Architecture

In this section, we provide an architectural overview of StegBench. StegBench is composed of three logically abstracted modules: dataset, embedder, and detector. Figure 4-1 shows which system requirements each module satisfies as well as which assets each module consumes and produces. These modules provide system modularity and high-usability as they effectively separate steganalysis workflows. In StegBench, all assets (steganographic embedders, steganalyzers, datasets) are referred to by a universally unique identifier (UUID).

4.3.1 Dataset Module

The dataset module is responsible for processing and downloading various image datasets to generate diverse cover datasets. By providing a highly configurable dataset

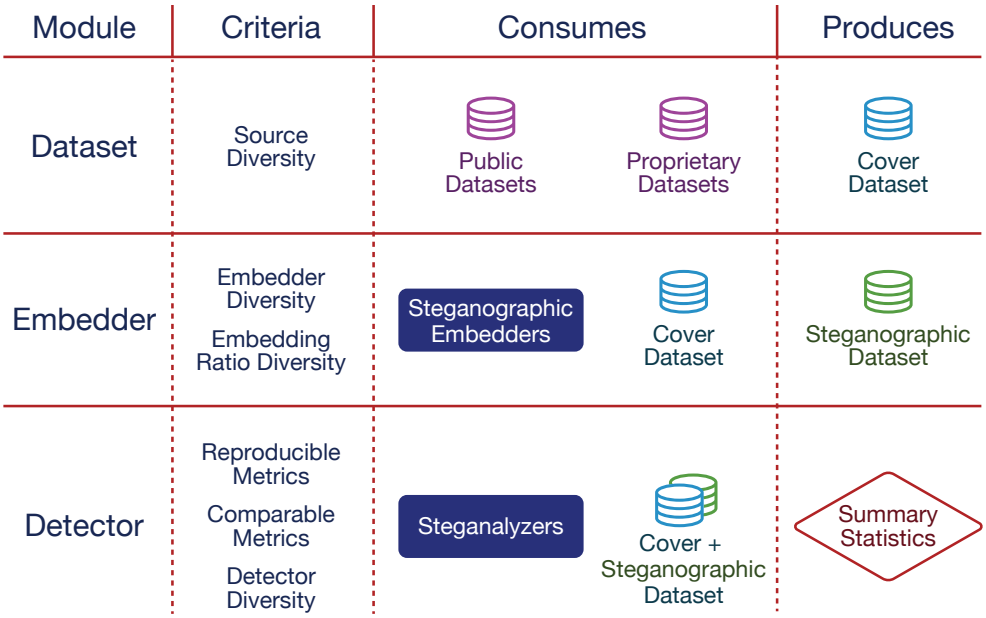

Figure 4-1: StegBench is divided into three modules: dataset, embedder, and detector. The figure shows what assets each module consumes and produces as well as system requirements that each module satisfies. The dataset module generates cover datasets. The embedder module generates steganographic datasets. The detector module evaluates steganalyzers on cover and steganographic datasets to produce summary statistics.

ALASKA	BOSS	BOWS2	MSCOCO	DIV2K

Table 4.1: List of public datasets that are supported by StegBench download routines.

management module, StegBench meets the source diversity requirement.

Figure 4-2 shows dataset module usage patterns using the StegBench API. In Table C.10, we list API specifications for the dataset module. The dataset module uses the following process flow, as shown in Figure 4-3:

1. **Load Image Datasets:** The dataset module can load datasets from either public or proprietary sources. StegBench provides download routines that can be found in the API specifications to access the public datasets listed in Table 4.1. For other datasets, the dataset must already exist on the local machine for the module to be able to load the dataset images into the StegBench system.

2. **Apply Dataset Setup Operations:** Once the images from the dataset are loaded into StegBench, the module applies any specified dataset setup oper-

```python
import stegbench as steg

"""StegBench provides the download command to download
public datasets. This command encompasses steps 1-4"""
dwld_db_uuid = steg.download(download_routine, db_name,
                    image_operations, setup_operations)

"""StegBench provides the process command to process
custom datasets. This command encompasses steps 1-4"""
local_db_uuid = steg.process(path_to_dir, db_name,
                    image_operations, setup_operations)
```

Figure 4-2: The dataset module is used to produce diverse cover datasets. Usage of the dataset module API involves loading or processing image datasets and applying image or dataset operations to produce a cover dataset.

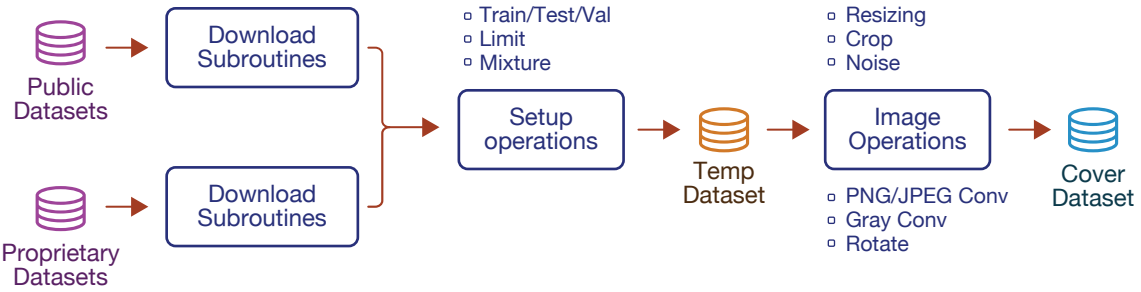

Figure 4-3: The dataset module generates cover datasets. The module provides subroutines to either download or load public/proprietary datasets. The module then applies any user-specified setup or image operations to produce diverse cover datasets. The dataset module provides a large set of modification operations and out-of-box access to several large datasets to generate robust cover datasets.

ations. These operations do not modify individual images but rather apply operations to the entire dataset (i.e. limiting the size of the dataset). The list of supported setup operations can be found in Table C.10. This step also generates a temporary dataset so that the original source dataset is not corrupted when further image operations are applied.

3. **Apply Image Operations:** If any image operations are specified, the module applies them efficiently to each image in the dataset. The list of supported image operations can be found in Table C.10. Since multiple operations can be applied at once, StegBench applies operations in a sequential manner to ensure

a deterministic output. Furthermore, all image operations are done in-place on the temporary dataset.

4. **Assign UUID and Extract Metadata:** Once image operations have completed, the module assigns the dataset a UUID. This UUID is used by all StegBench operations to refer to this specific cover dataset. Next, the module extracts and saves all relevant metadata from the dataset such as its UUID, the setup and image operations that were used for dataset generation, and the source of the dataset. The metadata is stored separately to enable quick and efficient lookup of dataset information.

4.3.2 Embedder Module

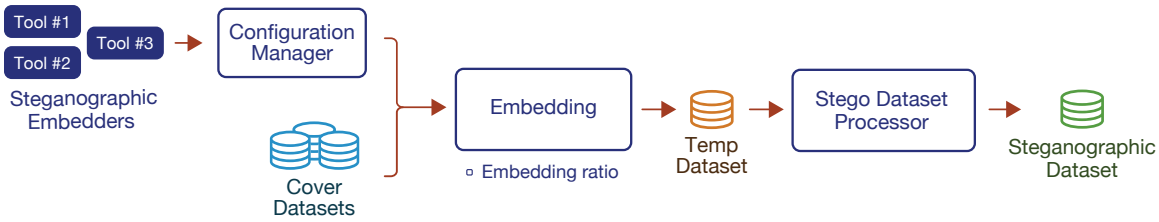

Figure 4-4: The embedder module generates steganographic datasets. Using the StegBench API, users can load steganographic embedders that are defined in configuration files via the configuration manager as well as specify any cover dataset(s). Next, using user-supplied embedding configurations, the module applies steganographic embedders to generate temporary steganographic images, which are then processed and combined into a steganographic dataset.

The embedder module is responsible for the generation of steganographic datasets. By providing a highly compatible and configurable embedder module, StegBench seamlessly integrates with a large set of steganographic embedders and effectively uses them with a wide range of embedding ratios, thereby satisfying the steganographic embedder diversity and embedding ratio diversity system criteria.

Figure 4-6 shows embedder module usage patterns using the StegBench API. In Table C.11, we list API specifications for the embedder module. The embedder module uses the following process flow, as shown in Figure 4-4:

```
embedder_cfg.ini

[steganogan]
algorithm_type = embeddor
compatible_types = ['png', 'pgm']
max_embedding_ratio = 4.0
command_type = native
run = steganogan encode INPUT_IMAGE_PATH SECRET_TXT -o OUTPUT_IMAGE_PATH
verify = steganogan decode INPUT_IMAGE_PATH
```

Figure 4-5: An example configuration for SteganoGAN. Steganographic embedder configurations specify compatibility requirements and embedding and decoding commands.

1. **Satisfy Prerequisites:** Steganographic embedders must already be installed on the local machine. For example, SteganoGAN[1] can be installed using pip[2]. Next, all cover datasets must already have been processed and assigned a UUID.

2. **Complete Setup Procedures**

   (a) **Load Configurations:** Once the steganographic embedders are installed, the user must create a configuration for each of them. Figure 4-5 shows an example configuration for SteganoGAN. Configurations detail compatibility requirements and execution specifications. For more details on configuration files, we refer the reader to Appendix C. The embedder module uses a configuration management system to load configuration files for steganographic embedders located on the user machine. During this process, each steganographic embedder is assigned a UUID that can be used to refer to it later on.

   (b) **Retrieve UUIDs:** Using the info command found in Table C.8, the user can retrieve UUIDs for steganographic embedders or cover datasets. For example, in Figure 4-6, the user uses the info command to retrieve the UUID for SteganoGAN and the BOSS dataset.

[1] https://github.com/DAI-Lab/SteganoGAN
[2] https://pip.pypa.io/

```
import stegbench as steg
"""Step 1: Satisfy Prerequisites"""
"""Step 2(a): Load Configuration"""
steg.add_config(config=['embeddor_cfg.ini'])
"""Steg 2(b): Retrieve UUIDs"""
steg.info()
>>> BOSS_COVER: uuid = '76e9535b-eabd'
>>> SteganoGAN: uuid = 'add568b9-a0a3'
steganogan_uuid = 'add568b9-a0a3'
boss_db_uuid = '76e9535b-eabd'

"""Step 3: Generate Steganographic Dataset"""
embedding_ratio = 0.5
stego_db_uuid = steg.embed(steganogan_uuid, boss_db_uuid,
                           embedding_ratio)

"""Step 4: Verify Steganographic Dataset"""
steg.verify(stego_db_uuid)
>>> Database: 100% correctly embedded
```

Figure 4-6: Step by step code usage patterns for the embedder module. The embedder module is used to generate and verify steganographic datasets. UUIDs are used to select the cover datasets and steganographic embedders used for embedding.

3. **Generate Steganographic Dataset**

   (a) **Apply Embedding Procedure:** Once steganographic embedders and cover datasets are selected using UUIDs, the embedder module uses the steganographic embedders to embed the cover dataset using a user-supplied embedding ratio. To do this, the module uses each steganographic embedder's configuration file to retrieve execution specifications. Then, using the execution specifications, the module generates executable commands that apply the steganographic embedder to the cover dataset. Finally, once all the commands have been generated, the embedder module executes the commands in parallel.

   (b) **Process Steganographic Dataset:** Once all the commands have finished executing, the embedder module: (1) cleans up any unnecessary files, (2) collects all the generated steganographic images into one directory, (3) assigns the steganographic dataset a UUID, and (4) extracts and

records all relevant metadata.

4. **Verify Steganographic Dataset:** For steganographic embedders that provide a method to decode steganographic images, StegBench enables the verification of steganographic datasets. To do this, StegBench uses steganographic decoder configurations to generate decoding commands. Once it applies these decoding commands, it verifies that the decoded steganographic content matches the data that was originally embedded into each steganographic image in the dataset.

4.3.3 Detector Module

Figure 4-7: The detector module evaluates steganalyzers across cover and steganographic datasets. Using the StegBench API, users can load steganalyzers that are defined in configuration files via the configuration manager as well as specify any cover or steganographic dataset(s). Next, the module evaluates each steganalyzer on the dataset images, collects these results, and uses analysis subroutines to properly generate summary statistics.

The detector module is responsible for the evaluation of steganalyzers. The module consumes steganalyzers along with cover and steganographic datasets to produce summary statistics. By providing seamless integration with a large array of steganalyzers and also by generating reproducible and comparable metrics, the detector module meets all evaluation criteria.

Figure 4-9 shows detector module usage patterns using the StegBench API. In Table C.12, we list API specifications for the detector module. The detector module uses the following process flow, as shown in Figure 4-7:

1. **Satisfy Prerequisites:** Steganalyzers must already be installed on the local machine. For example, StegExpose[3] can be installed from the source. Next, all

[3] `https://github.com/b3dk7/StegExpose`

```
detector_cfg.ini

[StegExpose]
algorithm_type = detector
compatible_types = ['png', 'bmp']
command_type = native
run = java -jar StegExpose.jar INPUT_IMAGE_DIRECTORY
```

Figure 4-8: An example configuration for StegExpose. Steganalyzer configurations specify compatibility requirements and detection commands.

datasets must already have been processed and assigned a UUID.

2. **Complete Setup Procedures**

   (a) **Load Configurations:** Once the steganalyzers are installed, the user must create a configuration for each of them. Figure 4-8 shows an example configuration for StegExpose. Configurations detail compatibility requirements and execution specifications. For more details on configuration files, we refer the reader to Appendix C. The detector module uses a configuration management system to load configuration files for steganalyzers located on the user machine. During this process, each steganalyzer is assigned a UUID that can be used to refer to it later on.

   (b) **Retrieve UUIDs:** Using the info command found in Table C.8, the user can retrieve UUIDs for steganalyzers or datasets. For example, in Figure 4-9, the user uses the info command to retrieve the UUID for StegExpose and a pair of cover and steganographic datasets.

3. **Test Steganalysis Performance**

   (a) **Apply Detection:** Once steganalyzers and datasets are selected using UUIDs, the detector module tests each steganalyzer on the datasets. To do this, the module uses each steganalyzer's configuration file to retrieve execution specifications. Then, using the execution specifications, the module

```
import stegbench as steg
"""Step 1: Satisfy Prerequisites"""
"""Step 2(a): Load Configuration"""
steg.add_config(config=['detector_cfg.ini'])
"""Steg 2(b): Retrieve UUIDs"""
steg.info()
>>> BOSS_COVER: uuid = '76e9535b-eabd'
>>> BOSS_STEGO: uuid = '991f73fa-44c1'
>>> StegExpose: uuid = 'add568b9-a0a3'
cover_db_uuid = '76e9535b-eabd'
stego_db_uuid = '991f73fa-44c1'
stegexpose_uuid = 'add568b9-a0a3'

"""Step 3: Test Steganalysis Performance"""
datasets_to_detect_uuids = [cover_db_uuid, stego_db_uuid]
steg.detect(stegexpose_uuid, datasets_to_detect_uuids)
>>> StegExpose: 85% accuracy, ...
```

Figure 4-9: Step by step code usage patterns for the detector module, which is used to measure steganalyzer performance across user-supplied datasets.

generates executable commands that have the steganalyzer detect whether a given image is steganographic. Finally, once all the commands have been generated, the module executes the commands in parallel.

(b) **Process Results:** The module collects the steganalysis results and calculates binary-classification metrics such as accuracy rates and F1 scores. The module then reports these summary statistics to the user.

Chapter 5

StegBench Experiments

In this chapter, we detail findings from several experiments carried out using the StegBench system and discuss which strategies are effective for building universal and practical steganalyzers. In section 5.1, we review our experimental setup. In section 5.2, we benchmark our architectures against state-of-the-art steganalyzers. In section 5.3, we conduct experiments to try to overcome the image size mismatch problem. In section 5.4, we test different solutions for the source mismatch problem. In section 5.5, we experiment with various training configurations to explore how we can solve the steganographic embedder mismatch problem. In section 5.6, we summarize our findings.

5.1 Experimental Setup

Each experiment follows a set of general specifications. If an experiment requires a unique configuration, we detail the specifications in the corresponding section. The following are the general specifications that we follow:

- **Source Specification:** For most experiments, we use 256x256 grayscale images from the COCO [22] dataset. Datasets are split into training, validation and test sets. Training and validation datasets are used to train deep learning models. The validation dataset can be switched with the training dataset for cross-validation. Test datasets are not seen during training and are used to test model

performance. For each experiment, we will lay out the source specifications. Please note that when 'x2' is used to refer to dataset size, it denotes that each image has a steganographic counterpart, thereby doubling the dataset size.

- **Steganographic Embedder Specification:** To simplify experiments, we use only spatial and deep learning steganographic embedders. For each experiment, we specify the steganographic embedders and embedding ratios used on the dataset. For SteganoGAN, we report Reed Solomon bpp [47], which is an equivalent measure to standard bpp.

- **Training Specification:** Models are trained for 100 epochs. Learning rate is set to 0.001 which decays once the loss plateaus for 10 epochs. We use a batch size of 32, which is composed of 16 cover images and 16 steganographic counterparts.

- **Software Specification:** StegBench is leveraged as an evaluation orchestration tool. PyTorch is used to implement deep learning models. Finally, we use the MATLAB implementations for DDE Lab tools (WOW, S_UNIWARD, HILL).

- **Hardware Specification:** We use an AWS Ubuntu Deep Learning AMI p2 machine with 8 Tesla K80 GPUs and 32 vCPUs.

5.1.1 Performance Measurement

We calculate the total detection error by setting an unoptimized threshold of 0.5 since model output is in the range: [0,1]. We use the following decision tree on the output of the deep learning steganalyzer to classify an input image:

- $< \mathbf{0.5}$: Normal

- $\geq \mathbf{0.5}$: Steganographic

Once these classifications are made, we calculate the total detection error, which is the proportion of incorrect classifications.

5.2 Benchmark

We first conduct an experiment to benchmark our steganalyzers, ArbNet and FastNet, against several state-of-the-art steganalyzers. We use the following procedure:

1. Train YeNet steganalyzer using the following datasets:
   - **Source Specification:** COCO: 9,000x2 training, 1,000x2 validation
   - **Steganographic Embedder Specification:** WOW at 0.1 bpp
2. There will be one YeNet steganalyzer at the end of step 1. Test it on the following dataset:
   - **Source Specification:** COCO: 5,000x2 test. The test dataset is generated with the same steganographic embedder: WOW at 0.1 bpp
3. Repeat steps 1-2 for $[0.2 \rightarrow 0.5]$ bpp.
4. Repeat steps 1-3 for steganographic embedders: S_UNIWARD and HILL.
5. Repeat steps 1-4 for XuNet, SRNet, ArbNet, and FastNet.
6. The resulting detection errors on the test sets can be found in Table B.1. The detection error trend lines are shown in Figure 5-1. The number of times a specific steganalyzer achieved the best performance for a given embedding ratio across the three steganographic embedders is reported in Table 5.1.

Using the results from the experiment, we make the following observations:

- **Trends agree with the literature** - We find good agreement between the trends in Figure 5-1 and the trends agreed upon in the literature [5, 41, 44]. Specifically, we find that YeNet and XuNet perform similarly across all steganographic embedders and embedding ratios. We also find that SRNet always has a lower detection error across these configurations. These trends match with those that have commonly been found in other steganalysis survey papers.

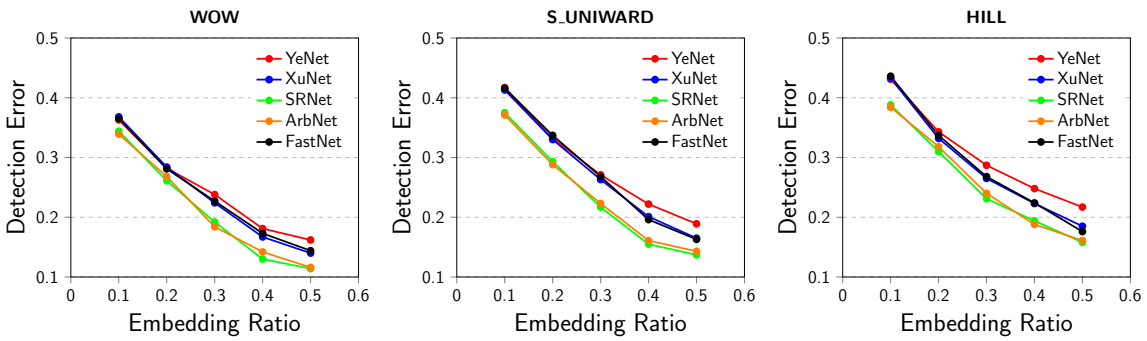

Figure 5-1: Detection error for five steganalyzers on test sets embedded with three different steganographic embedders. SRNet and ArbNet always perform the best across each test set configuration compared to the other steganalyzers. YeNet, XuNet, and FastNet all perform similarly, except for at higher embedding ratios where YeNet gets a slightly higher detection error.

Detectors	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**
YeNet	0	0	0	0	0
XuNet	0	0	0	0	0
SRNet	0	**2**	**2**	**2**	**3**
ArbNet	**3**	1	1	1	0
FastNet	0	0	0	0	0

Table 5.1: Number of wins across each of the three steganographic embedders (WOW, S_UNIWARD, HILL) for a given embedding ratio. Bolded numbers correspond to the steganalyzer that had the greatest number of wins for a given embedding ratio across the three steganographic embedders.

- **SRNet and ArbNet are the best performing steganalyzers** - SRNet and ArbNet are the best performing models across all steganographic embedders and embedding ratios. Compared to the other networks, SRNet and ArbNet both use skip connections and are able to more effectively compute residual noise signal.

- **ArbNet is good at low embedding ratios** - ArbNet shows stronger performance at lower embedding ratios. Specifically, in Table 5.1, we see that ArbNet shows its strongest performance at 0.1 bpp. This suggests that using techniques found in ArbNet could help solve the low embedding ratio problem. More research is needed to identify why these techniques are effective here.

- **FastNet performs comparably to other steganalyzers** - FastNet performs

comparably to other steganalyzers. FastNet is based on the EfficientNet architecture, which was designed specifically for the practical need of training and execution efficiency. Thus, this result shows a promising sign that FastNet could be a way to generate a steganalyzer that meets this practical need.

5.3 Image Size Mismatch Problem

Problem Definition: Most steganalyzers are limited to only detecting images of a specific size. Those that do detect arbitrary image sizes use techniques such as global average pooling that lead to less accurate results.

Proposed Solution: Train a steganalyzer which has a spatial pyramid pooling layer such as ArbNet on multiple image sizes.

Experimental Procedure: To test this solution, we use the following steps:

1. Train SRNet and YeNet on the following datasets:
    - **Source Specification:** COCO(256x256): 9,000x2 training, 1,000x2 validation
    - **Steganographic Embedder Specification:** WOW at 0.5 bpp
2. Repeat step 1 for the following image sizes: 512x512 and 1024x1024
3. Train ArbNet on the following dataset:
    - **Source Specification:** COCO(256x256,512x512, and 1024x1024): 9,000x2 training, 1,000x2 validation
    - **Steganographic Embedder Specification:** WOW at 0.5 bpp
4. There will be three SRNet, three YeNet, and one ArbNet steganalyzer at the end of steps 1-3. Test each one on the following datasets:
    - **Source Specification:** COCO(256x256, 512x512, or 1024x1024): 5,000x2 test. The test dataset is generated with the same steganographic embedder: WOW at 0.5 bpp.

5. Repeat steps 1-4 for steganographic embedder HILL at 0.5 bpp

6. The resulting detection errors can be found in Table B.2.

7. For each steganalyzer, we subtract the 256x256 test set detection error from the 1024x1024 test set detection error. This number is reported in Figure 5-2.

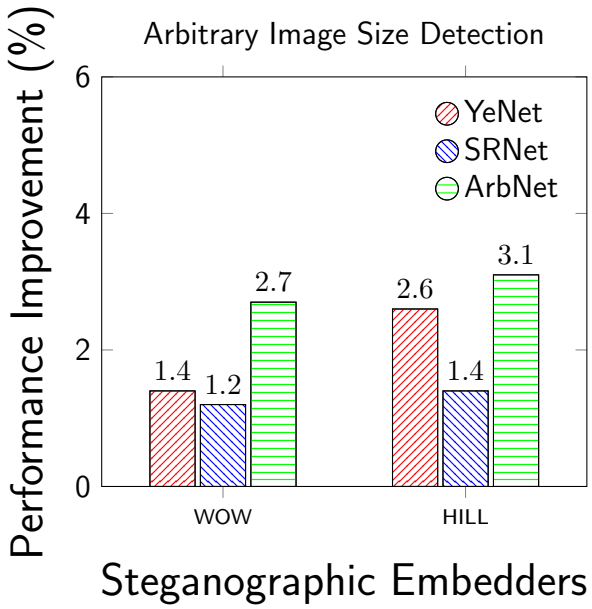

Figure 5-2: Performance gain from 256x256 to 1024x1024 across three different steganalyzers for two steganographic embedders (WOW, HILL). The performance gain is the difference in detection error between these two test sets. Across the board, steganalyzers improved in performance when detecting images of higher sizes.

Using the results from the experiment, we make the following observations:

- **Steganalyzer performance improves as image size increases** - In Figure 5-2, we see that for both WOW and HILL, all three steganalyzers improve in performance when tested on the 1024x1024 test set compared to the 256x256 test set. We believe that larger image sizes allow steganalyzers to capture larger amounts of steganographic signal, which makes for easier steganalysis detection.

- **ArbNet is an effective solution for the image size mismatch problem** - We find that ArbNet can be robustly applied to various image sizes, making it an effective solution. Specifically, ArbNet is able to perform effectively

across all image sizes even though it was trained on a composite of image sizes. Furthermore, ArbNet shows better performance gains than other steganalyzers on larger image sizes. This suggests that using a spatial pyramid pooling layer like we did in ArbNet is an effective way to avoid downsampling steganographic signal when detecting larger images.

5.4 Source Mismatch Problem

Problem Definition: Steganalyzers perform poorly on datasets that come from a different source than the dataset used to train the steganalyzers.
Proposed Solution: Increase source diversity in the training dataset.
Experimental Procedure: To test this solution, we use the following steps:

1. Train SRNet steganalyzers on the following datasets:
    - **Source Specification:** COCO, BOSS, COCO + BOSS: 9,000x2 training, 1,000x2 validation
    - **Steganographic Embedder Specification:** WOW at 0.5 bpp
2. There will be three SRNet steganalyzers at the end of step 1. Test each one on the following datasets:
    - **Source Specification:** COCO: 5,000x2 test. BOSS: 5,000x2 test. The test datasets are generated using the same steganographic embedder: WOW at 0.5 bpp.
3. Repeat steps 1 and 2 for steganographic embedder HILL at 0.5 bpp.
4. The detection errors on the test datasets are listed in Table B.3 and graphically displayed in Figure 5-3.
5. Using the detection errors, we calculate a customized metric known as the source mismatch metric, shown in Table 5.2, using the following steps:

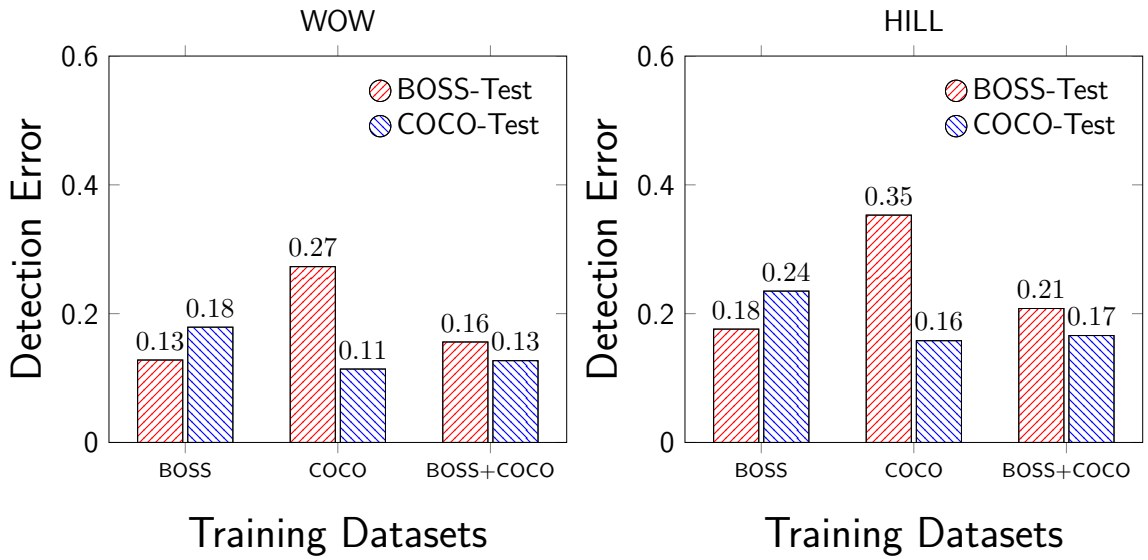

Figure 5-3: Detection error for SRNet when trained on either BOSS, COCO, or BOSS+COCO and tested on either BOSS or COCO. The left plot shows the detection error for datasets embedded with WOW and the right plot shows detection error for datasets embedded with HILL.

(a) Collect all detection errors for steganalyzers tested on the COCO dataset and either trained on BOSS or COCO.

(b) Take the difference in the detection error between the BOSS-trained and COCO-trained steganalyzers for WOW. In this case, this would be 0.18 (BOSS) - 0.11 (COCO) = 0.07.

(c) Take the difference in the detection error between the BOSS-trained and COCO-trained steganalyzers for HILL. In this case, this would be 0.24 (BOSS) - 0.16 (COCO) = 0.08.

(d) Take the average of the two differences ($\frac{0.08+0.07}{2} = 0.075$). This is the source mismatch metric for BOSS-trained steganalyzers.

(e) Repeat steps 1-4 for steganalyzers tested on the BOSS dataset.

Using the results from the experiment, we make the following observations:

- **Steganalyzers perform best when the training and test dataset sources match** - As seen in both plots of Figure 5-3, the best detection error is always

Train Set	Source Mismatch Metric
BOSS	0.075
COCO	0.155

Table 5.2: The source mismatch metric is the average increase in detection error for a steganalyzer trained on a dataset, D, compared to a steganalyzer trained on a dataset, $D^{'}$, when both are tested on $D^{'}$. In this table, we show the comparisons between BOSS and COCO. The metric shows how effective each source is for training when tested on another source. A lower metric indicates that the training dataset is better for overcoming the source mismatch problem.

achieved when the training and test datasets come from the same source. For example, for WOW, the lowest detection error is achieved by SRNet trained on COCO and tested on COCO. We expect this result since steganalyzers trained and tested on the same dataset source can exploit source distribution signals.

- **All steganalyzers trained and tested on different dataset sources suffer from the source mismatch problem** - In both plots of Figure 5-3, we see that steganalyzers that are trained and tested on different dataset sources perform poorly. For example, SRNet trained on COCO using HILL achieves a detection error of 0.16 on a COCO test set, while it achieves a fairly poor detection error of 0.35 on a BOSS test set.

- **BOSS-trained steganalyzers overcome the source mismatch problem more effectively than COCO-trained steganalyzers** - Using the source mismatch metrics from Table 5.2, we see that BOSS-trained steganalyzers achieve a metric of 0.075 while COCO-trained steganalyzers achieve a metric of 0.155. A lower source mismatch metric indicates that the training dataset helps a steganalyzer better overcome the source mismatch problem. We believe that this result is likely because the BOSS dataset is more 'textured' (has more high-frequency signal) and presents a harder steganalysis challenge [49].

- **Increasing source diversity is useful in limited contexts** - For steganalyzers trained on the BOSS+COCO dataset, we see that the detection error on BOSS and COCO decreased compared to just being trained on either BOSS or

COCO. Yet, steganalyzers that were trained on both datasets performed worse compared to the matching train/test situations (i.e. COCO-train/COCO-test, BOSS-train/BOSS-test). Still, to conclusively identify the effects of increasing source diversity, we will need to test all steganalyzers on a third independent dataset, which we have not done in this thesis.

5.5 Steganographic Embedder Mismatch Problem

In this section, we experiment with several proposed solutions to the steganographic embedder mismatch problem. We define the problem as follows:
Problem Definition: Steganalyzers tend to fail at detecting steganographic images that are created from a steganographic embedder that they have not been trained on.

5.5.1 Single Steganographic Embedder

Proposed Solution: Use deep learning steganographic embedders such as SteganoGAN to create training datasets.
Experimental Procedure: To test this solution, we use the following steps:

1. Train SRNet steganalyzer on the following datasets:
    - **Source Specification:** COCO; 9,000x2 training, 1,000x2 validation
    - **Steganographic Embedder Specification:** WOW at 0.5 bpp.
2. Repeat step 1 for steganographic embedders: S_UNIWARD, HILL, and SteganoGAN at 0.5 bpp.
3. There will be four SRNet steganalyzers at the end of steps 1 and 2. Test each one on the following datasets:
    - **Source Specification:** COCO; 5,000x2 test.
    - **Steganographic Embedder Specification:** Each steganalyzer is tested on all the following steganographic embedders: WOW, S_UNIWARD, HILL, or SteganoGAN at 0.5 bpp

4. The resulting detection errors are listed in Table B.4 and graphically shown in Figure 5-4.

5. For each steganographic embedder, we take the difference between the mismatched-scenario detection error and the matching-scenario detection error to get the relative increase in detection error when the train/test steganographic embedders are mismatched. These metrics are plotted in Figure 5-5.

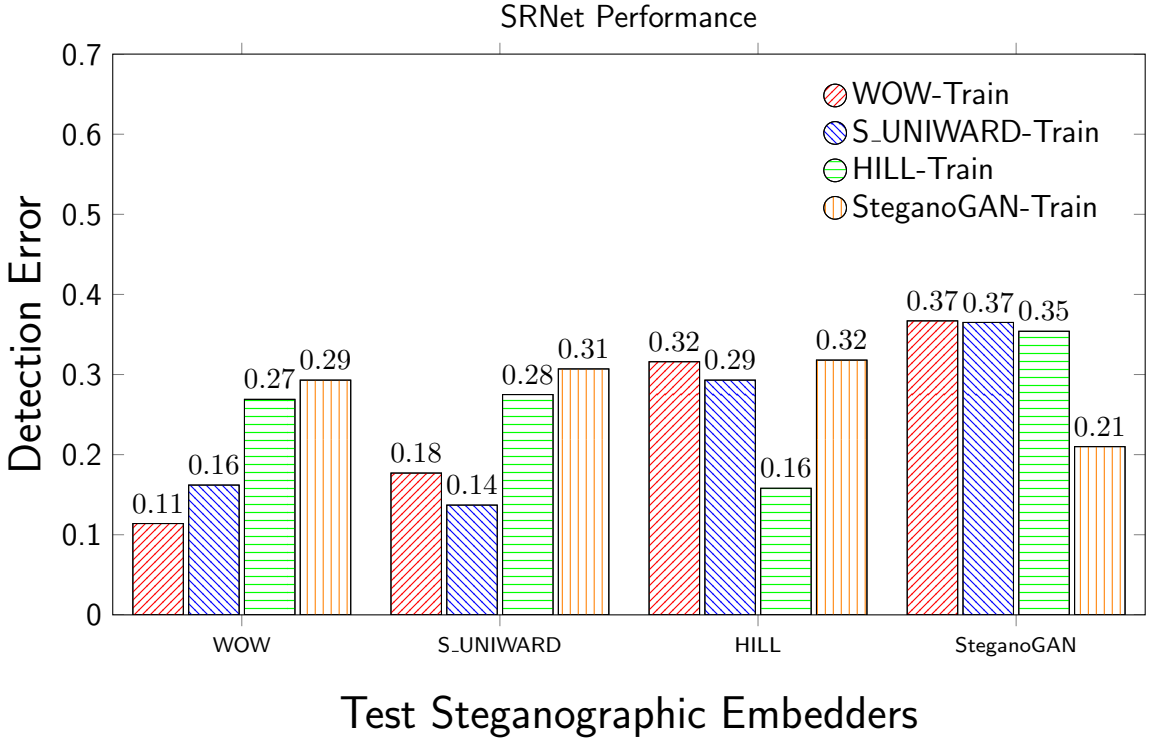

Figure 5-4: Total detection error for steganalyzers trained on the steganographic embedder specified by the legend and tested on the steganographic embedder specified by the x-axis. In all test situations, the lowest detection error was achieved by steganalyzers trained on the same steganographic embedder. SteganoGAN was by far the hardest steganographic embedder to detect.

Using the results from the experiment, we make the following observations:

- **Steganalyzers achieve the highest performance when the training and test steganographic embedder are the same** - In Figure 5-4, we see that for each of the test sets, the lowest detection error is always achieved when the training/test steganographic embedder are the same. For example, on the

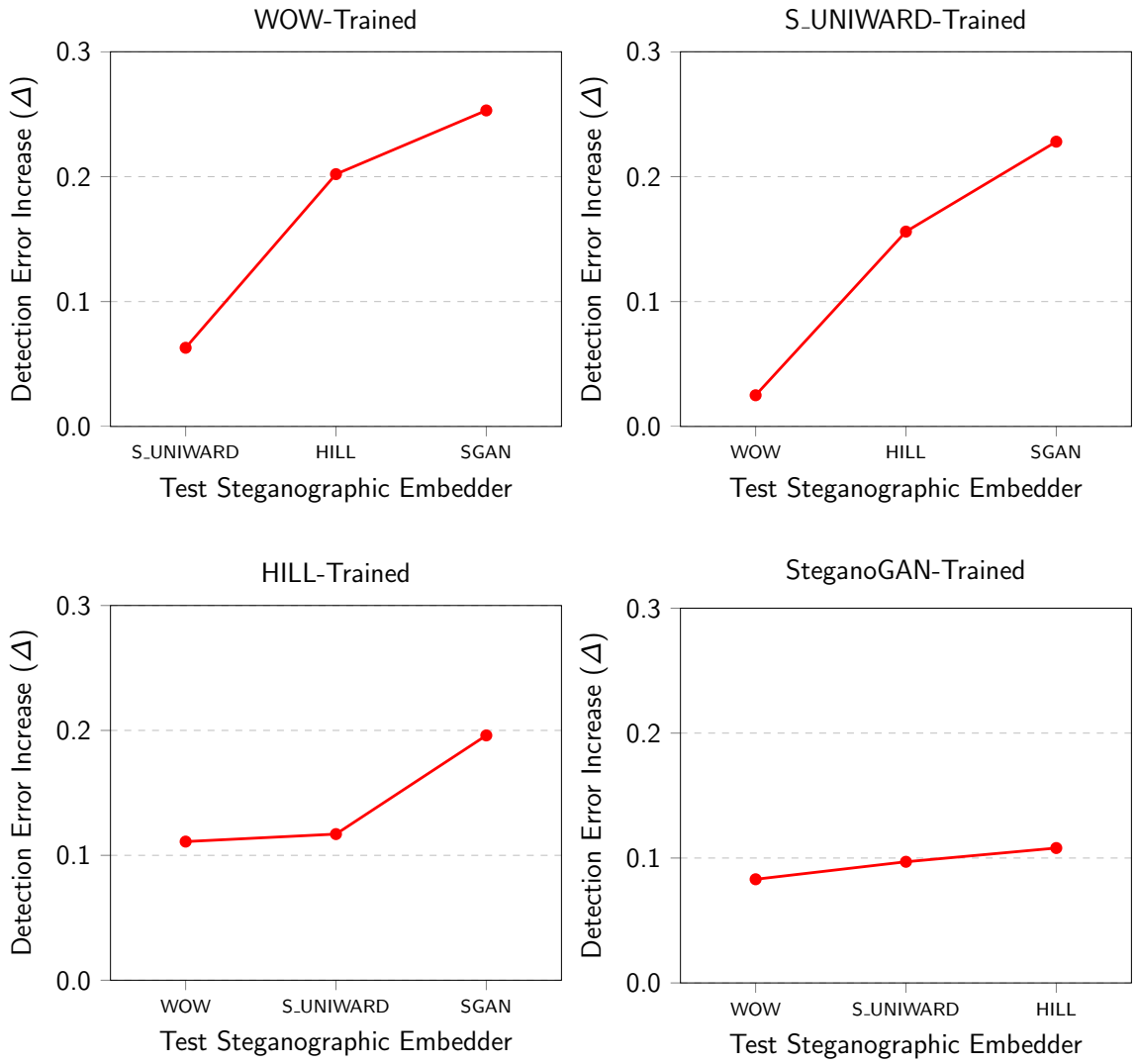

Figure 5-5: The relative increase in detection error for SRNet during the mismatch steganographic embedder test scenario compared to the matching scenario.

WOW test set, the lowest detection error of 0.11 is achieved by a WOW-trained steganalyzer. We expect this result since deep learning steganalyzers are optimized for the steganographic embedder that they are trained on.

- **Steganalyzers achieve good performance when the training and test steganographic embedder are similar** - In Figure 5-4 and Figure 5-5, we see that WOW-trained steganalyzers perform reasonably well on S_UNIWARD test sets and vice versa. Specifically, these steganalyzers achieve comparable detection errors. For example, on the WOW test set, the S_UNIWARD-trained

steganalyzer achieves a detection error only 0.05 worse than the WOW-trained steganalyzer. We believe this is because WOW and S_UNIWARD use similar distortion functions to embed steganographic content into images. Thus, steganalyzers are able to accurately detect steganographic images from non-trained steganographic embedders when the trained and non-trained steganographic embedders work similarly.

- **Steganographic images from SteganoGAN are the hardest to detect** - In Figure 5-4 and Figure 5-5, we can see that for the train/test mismatch situation, testing on SteganoGAN always produces the highest total detection error and consequently the highest increase in detection error. For example, steganalyzers trained on WOW, HILL, or S_UNIWARD show an increase of at least 0.2 in detection error when detecting steganographic images from SteganoGAN. This shows that SteganoGAN is the hardest steganographic embedder to detect when it has not been trained on. Furthermore, it shows that SteganoGAN embeds steganographic signal in a more secure manner, as steganographic signals from other steganographic embedders are not effective for classifying steganographic images generated by SteganoGAN.

- **Steganalyzers trained on SteganoGAN seem to be better at overcoming the steganographic embedder mismatch problem** - We find that steganalyzers trained on SteganoGAN do not show significant increases in detection error when detecting steganographic images from other steganographic embedders. Specifically, in the fourth plot of Figure 5-5, we see that detection error does not increase much against other steganographic embedders. In this plot, we see a relatively flat slope compared to the other plots which indicates that SteganoGAN training datasets are useful at learning other steganographic signals. Still, when looking at Figure 5-4, we see that SteganoGAN-trained steganalyzers get the highest detection error on all other steganographic embedders. More research is needed to understand the effectiveness of SteganoGAN training examples for the steganographic embedder mismatch problem.

5.5.2 Multiple Steganographic Embedders

Proposed Solution: Use multiple steganographic embedders to create training datasets.

Experimental Procedure: To test this solution, we use the following steps:

1. Train SRNet steganalyzers on the following dataset:
   - **Source Specification:** COCO; 9,000x2 training, 1,000x2 validation
   - **Steganographic Embedder Specification:** WOW + HILL
2. Repeat step 1 for steganographic embedder: SteganoGAN + SteganoGAN at 0.5 bpp.
3. There will be two SRNet steganalyzers at the end of steps 1 and 2. Test each one on the following datasets:
   - **Source Specification:** COCO; 5,000x2 test
   - **Steganographic Embedder Specification:** Each steganalyzer is tested on all the following steganographic embedders: WOW, HILL, or SteganoGAN at 0.5 bpp
4. The resulting detection errors are listed in Table B.5.
5. We graphically compare the results to the single steganographic embedder results from section 5.5.1 in Figure 5-6.

Using the results from the experiment, we make the following observations:

- **Mixing spatial steganographic embedders mitigates the steganographic embedder mismatch problem in limited contexts** - In the left plot of Figure 5-6, we see how the WOW+HILL-trained steganalyzer compares to the WOW-trained steganalyzer. Increasing the spatial steganographic embedder diversity did allow the steganalyzer to effectively reduce detection error on HILL but has relatively little impact on reducing detection error on SteganoGAN.

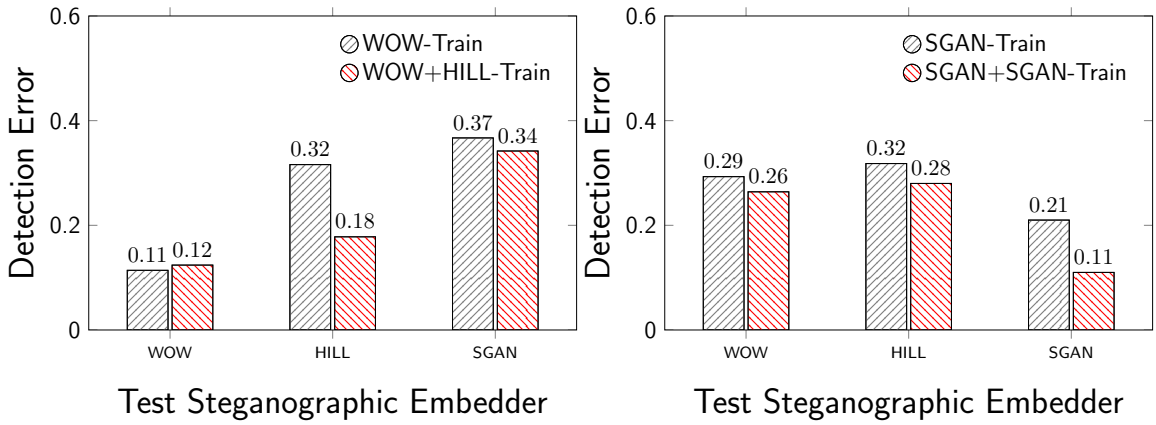

Figure 5-6: The gray bars show the detection error of SRNet trained on a single steganographic embedder while the red bars show the detection error of SRNet trained on multiple steganographic embedders. The detection error is calculated on a COCO dataset embedded by the steganographic embedder labeled on the x-axis.

Thus, increasing spatial steganographic embedder diversity only shows improvement against traditional steganographic embedders.

- **Mixing deep learning steganographic embedders mitigates the steganographic embedder mismatch problem in all contexts** - In the right plot of Figure 5-6, we see that increasing SteganoGAN instances in the training dataset reduces detection error in all contexts. This finding further emphasizes the effectiveness of effectiveness of SteganoGAN training examples in helping train a universal steganalyzer.

5.6 Summary

1. **Training and Execution Efficiency Problem** - FastNet provides a promising solution to improve efficiency since it achieves relatively accurate detection results using an EfficientNet architecture.

2. **Low Embedding Ratio Problem** - DenseNet architectures seem to be more effective than other techniques at detecting lower embedding ratios since they more effectively boost steganographic signal.

3. **Image Size Mismatch Problem** - Spatial pyramid pooling layers in deep learning architectures (i.e. ArbNet) enable and improve accuracy on arbitrary image size detection.

4. **Source Mismatch Problem** - Source diversity in training datasets with a preference for textured datasets like BOSS helps slightly mitigate this problem but also leads to decreases in performance.

5. **Steganographic Embedder Mismatch Problem** - Training datasets generated by multiple deep learning steganographic embedders such as SteganoGAN help steganalyzers better overcome this problem.

Chapter 6

Adversarial Attacks on Steganalyzers

In the previous chapters, we focused on identifying failure modes common to steganalysis and building robust deep learning steganalyzers. However, we have not yet discussed specific issues relevant to machine learning. Specifically, machine learning models are particularly prone to a class of attacks known as adversarial attacks. In these attacks, small changes known as adversarial perturbations are added to the input data to create an adversarial example. These small changes are used to deliberately fool a machine learning system into misclassifying the adversarial example. There are several adversarial attack scenarios that can be used to generate these adversarial perturbations. An exhaustive summary of different adversarial attack scenarios can be found in [19]. We summarize two of them below:

- **White Box** - These adversarial attacks are able to use full knowledge of the model including model type, model architecture, and values of all parameters and weights to generate adversarial perturbations.
- **Black Box** - These adversarial attacks must use limited to no knowledge about the model (for example, only the model outcome) to generate adversarial perturbations.

In this chapter, we only focus on the white box scenario since it is the most com-

monly researched attack scenario [19]. In this scenario, the most common methods for generating adversarial perturbations use the gradient of the model's loss function. Two gradient-based methods are the fast gradient sign method (FGSM) [13] and the projected gradient descent (PGD) method [24]. For additional detail on the exact mechanics of these gradient-descent methods, we refer the reader to section 2.1.3.

Since white-box adversarial attacks are able to effectively create adversarial examples that fool deep learning models, we believe it is important to examine how they can be used to fool a deep learning steganalyzer. Specifically, we aim to understand if deep learning steganalyzers are also prone to these attacks and if so, what preventive measures can be used to defend against them.

Thus, we must first explore how deep learning steganalyzers can be attacked using a white-box adversarial approach. In section 6.1, we explain the design of our attack system, StegAttack. In section 6.2, we conduct experiments to test the effectiveness of StegAttack. Finally, in section 6.3, we study how adversarial training procedures can be used to create universal and robust deep learning steganalyzers.

6.1 StegAttack System Design

6.1.1 Goals and Definitions

Using a white-box attack model, StegAttack aims to generate an adversarial steganographic image using a normal steganographic image. An adversarial steganographic image is a special type of steganographic image that meets the following conditions for a given deep learning steganalyzer:

- **[Condition 1] Detected by Steganalyzer:** It can be detected by the steganalyzer without the adversarial perturbation.
- **[Condition 2] Fools Steganalyzer:** It cannot be detected by the steganalyzer with the adversarial perturbation.
- **[Condition 3] Decodable:** It must be decodable.

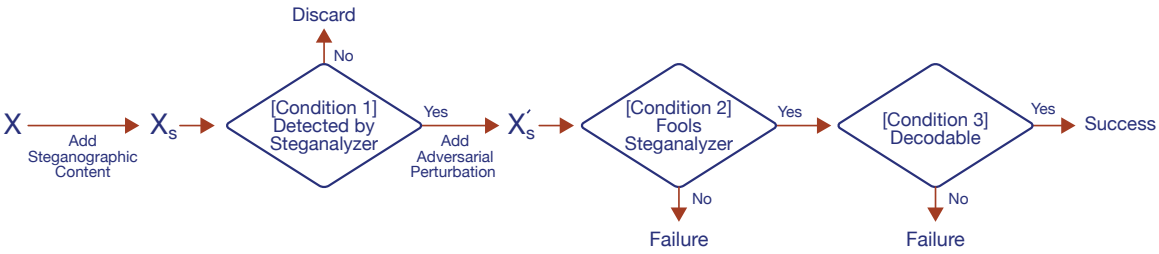

Figure 6-1: StegAttack V1 uses the process flow shown in this figure to introduce adversarial perturbations to a steganographic image to try to generate an adversarial steganographic image. V1 first check that a steganographic image, X_S, can already be detected by a steganalyzer. It then introduces adversarial perturbations to create X'_S. If X'_S can fool the steganalyzer and is still decodable, the attack is a success.

6.1.2 StegAttack V1

In this subsection, we describe the first version of StegAttack, shown in Figure 6-1. In this version, we use the following procedures:

1. **Adding Steganographic Content** - We first generate a normal steganographic image that can be used by StegAttack. To do this, we add steganographic content to an image, X, to create a steganographic image, X_S

2. **[Condition 1] Detected by Steganalyzer** - The first condition of an adversarial steganographic image requires that the steganographic image is detectable by the steganalyzer prior to the addition of any adversarial perturbation. If the steganographic image, X_S, is already able to avoid detection, the attack is discarded, since the steganographic image does not meet this condition and there is no need to add any perturbation to it.

3. **Generating and Adding Adversarial Perturbations** - For a steganographic image, X_S, that can be detected by the steganalyzer, we add adversarial perturbations. To generate the adversarial perturbation, StegAttack uses a gradient-based method (for instance, FGSM). The adversarial perturbation is then added to the steganographic image, X_S, to create X'_S

4. **[Condition 2] Fools Steganalyzer** - Once the adversarial perturbation is added, StegAttack must check if the steganographic image, X'_S, can fool the

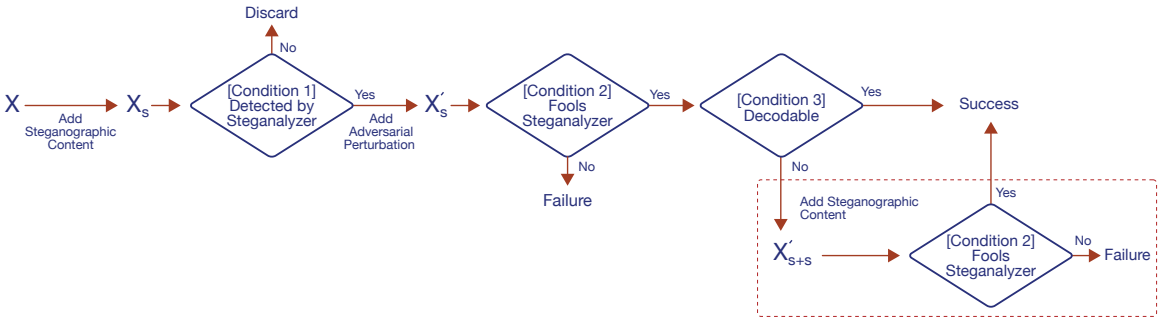

Figure 6-2: StegAttack V2, shown in the boxed area, expands on V1 by adding steganographic content to non-decodable steganographic images that can already fool the steganalyzer to try to create additional adversarial steganographic images. StegAttack V2 ensures that all conditions for an adversarial steganographic image still hold by checking if the re-embedded image, X'_{S+S} can still fool the steganalyzer.

steganalyzer. In this context, fooling the steganalyzer means that the steganalyzer classifies X'_S as a normal image. If the steganographic image X'_S cannot fool the steganalyzer, the attack fails, since X'_S does not meet the second conditional requirement of an adversarial steganographic image.

5. **[Condition 3] Decodable** - The steganographic image, X'_S, must still be decodable to be considered a steganographic image. This step is necessary since adversarial perturbations might corrupt the content in the steganographic image, thereby making the image undecodable. If the steganographic image is undecodable, the steganographic image is considered a normal image and does not meet the third condition for an adversarial steganographic image. However, if the steganographic image is decodable, then X'_S is an adversarial steganographic image.

6.1.3 StegAttack V2

In the previous section, we used adversarial perturbations to generate adversarial steganographic images. However, we may have been too quick to discard the non-decodable X'_S - after all, this image has already passed two of the three conditions required for it to be an adversarial steganographic image. Thus, in this next iteration of StegAttack, we focus on a procedure that tries to modify any non-decodable X'_S

such that it satisfies the third condition. By doing this, we improve StegAttack's effectiveness at generating adversarial steganographic images.

As shown in the boxed area of Figure 6-2, we use the following procedure to complete the StegAttack system:

1. **StegAttack V1** - We follow all steps included in the V1 system. These steps must still be followed to generate the adversarially-perturbed image, X'_S. In this part of StegAttack, we capture any image, X'_S, that could not be decoded but did meet the other two conditions for an adversarial steganographic image.

2. [**Condition 3**] **Decodable** - To satisfy condition three, we add steganographic content to X'_S to generate X'_{S+S}. By adding the steganographic content again, we can be sure that the steganographic image, X'_{S+S}, is decodable. But by doing this, we might also accidentally invalidate condition two since X'_{S+S} might not be able to fool the steganalyzer anymore.

3. [**Condition 2**] **Fools Steganalyzer** - To verify that condition two still holds, StegAttack checks whether X'_{S+S} can fool the steganalyzer. If it is able to fool the steganalyzer, X'_{S+S} is considered an adversarial steganographic image. Otherwise, the attack fails since the steganographic image, X'_{S+S}, does not meet condition two.

6.1.4 StegAttack Process Flow

The entire StegAttack process flow, shown in Figure 6-2, can be described using the following steps:

1. **Action:** Add steganographic content to an image X to produce X_S.

2. **Decision:** DETECTABLE(X_S, Steganalyzer):

    2.1. **Result: Yes -** The attack proceeds.

    2.2. **Result: No -** The attack is discarded.

3. **Action:** Generate and add adversarial perturbations to X_S to produce $X_S^{'}$ using a gradient-based method.

4. **Decision:** FOOLS($X_S^{'}$, Steganalyzer):

   4.1. **Result: Yes -** The attack proceeds.

   4.2. **Result: No -** The attack fails.

5. **Decision:** DECODABLE($X_S^{'}$)

   5.1. **Result: Yes -** The attack succeeds.

   5.2. **Result: No -** The attack proceeds.

6. **Action:** Embed $X_S^{'}$ with the same steganographic content from step 1, to produce $X_{S+S}^{'}$.

7. **Decision:** FOOLS($X_{S+S}^{'}$, Steganalyzer)

   7.1. **Result: Yes -** The attack succeeds.

   7.2. **Result: No -** The attack fails.

6.2 StegAttack Effectiveness

In this section, we test the effectiveness of StegAttack on different steganalyzers. In section 6.2.1, we show example adversarial steganographic images. In section 6.2.2, we detail the experiment setup that is common to each of the experiments conducted in this section. In section 6.2.3, we test how effective StegAttack is compared to a naive method. In section 6.2.4 we test how different gradient-descent methods affect StegAttack effectiveness.

6.2.1 Example Adversarial Steganographic Images

Figure 6-3 shows example adversarial steganographic images generated using StegAttack with FGSM as its gradient-based method. We note that this attack did not

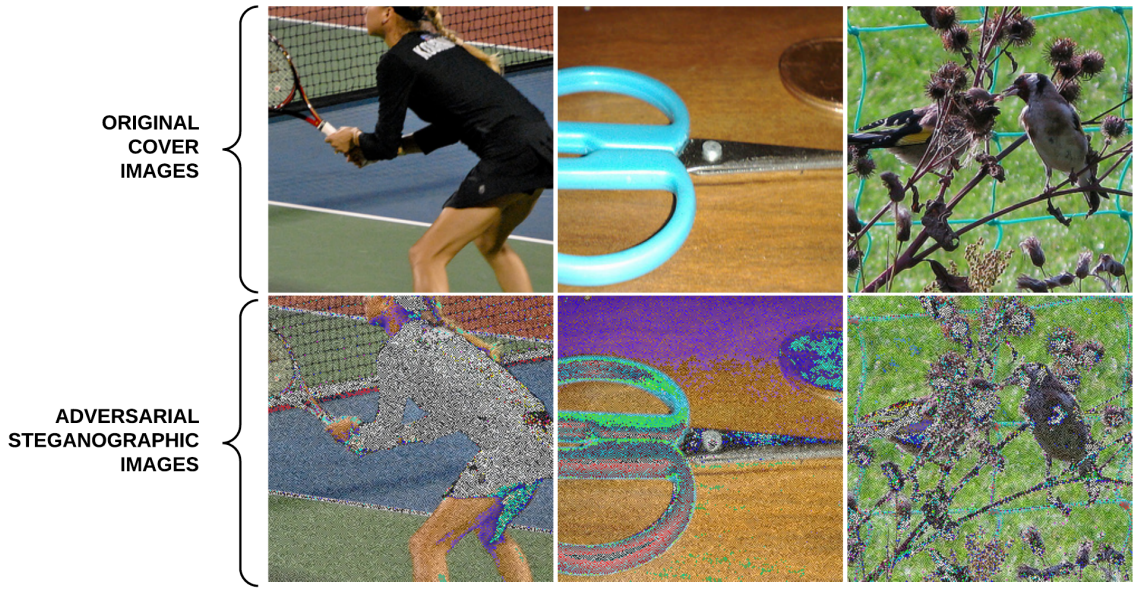

Figure 6-3: Adversarial steganographic images that are generated using StegAttack with FGSM($\epsilon = 0.3$). The adversarial steganographic image image quality is low because the images are generated using a heavy attack that modifies significant image content. Reducing the step size will enable better image quality but reduce StegAttack efficacy.

create high quality images. This is likely because we configure FGSM with a step size of $\epsilon = 0.3$, which ends up heavily modifying the steganographic image. We are quite confident that image quality can be improved by reducing the step size to reduce the effect of the perturbation. It is important to note that reducing the step size will also likely reduce StegAttack's effectiveness.

6.2.2 Experiment Setup

For the experiments in section 6.2.3 and section 6.2.4, we follow the same setup procedures described here. We use the following steps:

1. Train four steganalyzers (XuNet, YeNet, ArbNet, SRNet) using a COCO dataset of 9,000x2 training, 1,000x2 validation images embedded with LSB at 0.5 bpp

2. Collect a test set of 2,000 other steganographic images using the COCO dataset and LSB at 0.5 bpp. All images in the test set must meet condition 1 of

Steganalyzer	Naive Attack	StegAttack
XuNet	0%	**75.2%**
YeNet	0%	**81.3%**
ArbNet	0%	**6.2%**
SRNet	0%	**4.8%**

Table 6.1: Missed detection probability (P_{MD}) on four steganalyzers using a steganographic image test set with either a naive Gaussian-based attack or StegAttack.

StegAttack (they already detectable) so that results are not biased by inherent steganalyzer inaccuracies.

3. For each attack, the input steganographic image data is normalized to [0,1]. Attack parameters are scaled accordingly.

4. Each attack is then applied to the four steganalyzers using the test steganographic image dataset.

5. Upon attack completion, we record the percentage of adversarial steganographic images that each attack method generated. This measurement corresponds to each steganalyzer's missed detection probability, P_{MD}.

6.2.3 Effectiveness Compared to Naive Method

In this experiment, we compare the effectiveness between StegAttack and a naive Gaussian-based attack method against four different steganalyzers. We use the following attack method specifications:

- **StegAttack** - We use the gradient-descent method, FGSM, with a step size of $\epsilon = 0.3$ to generate the adversarial perturbation.

- **Naive Attack** - The naive attack method uses the exact same attack process flow as StegAttack but generates the adversarial perturbation by sampling Gaussian noise w/ $\sigma = 0.5$.

Table 6.1 reports results from our effectiveness experiment. Using this data, we make the following observations:

- **Naive Gaussian-based attack method is ineffective** - We find that the Gaussian-based attack method is unable to produce any adversarial steganographic images. This result makes sense since the adversarial perturbation used by the Gaussian-based attack method is nothing more than random noise that does not introduce significant confounding signals. The random noise usually ends up simply corrupting the steganographic signal even when the perturbed image does fool the steganalyzer. Furthermore, re-embedding images that do fool the steganalyzer end up making them detectable again.

- **StegAttack generates adversarial steganographic images for all steganalyzers** - For every steganalyzer, StegAttack was able to generate some amount of adversarial steganographic images.

- **StegAttack is less effective on certain steganalyzers** - StegAttack's effectiveness is not consistent across steganalyzers. Specifically, StegAttack is highly effective on XuNet and YeNet but not as much on ArbNet and SRNet. We find that more robust steganalyzers tend to have the following architectural similarities: (1) they disable pooling in the front part of the network; (2) they use skip connections and other methods to better computes noise residuals; and (3) they are deeper networks that more effectively extract steganographic signal. Each of these features helps these steganalyzers avoid downsampling input data, which might help them be more robust against StegAttack.

6.2.4 Effectiveness of Different Gradient-Descent Methods

In this experiment, we test how using different gradient-based methods impact the effectiveness of StegAttack. We use the following gradient-based methods:

- **FGSM w/ $\epsilon = 0.3$**

- **PGD w/ 40 iterations, $\epsilon = 0.3$**

Table 6.2 lists the results from this experiment. Using these results, we make the following observations:

Steganalyzer	FGSM	PGD
XuNet	75.2%	**98.2%**
YeNet	81.3%	**97.8%**
ArbNet	6.2%	**27.4%**
SRNet	4.8%	**15.2%**

Table 6.2: Missed detection probability (P_{MD}) on four steganalyzers using a steganographic image test set with one of two gradient-descent methods for StegAttack.

- **Stronger gradient-descent methods are more successful** - As seen in Table 6.2, PGD is more effective than FGSM at generating adversarial steganographic images.

- **StegAttack is less effective on certain steganalyzers even with better gradient-descent methods** - In Table 6.2, we see that even though PGD produces more adversarial steganographic images, it still not very effective against ArbNet and SRNet.

6.3 Adversarial Training

We now explore how adversarial training affects steganalyzer performance. Adversarial training is the use of adversarial images as part of a training set to increase model robustness [19]. In traditional adversarial training methods, the adversarial examples are generated during training time and included as part of a modified loss function.

However, in our setup, we cannot use StegAttack to generate adversarial examples during training time. This is because StegAttack requires the steganalyzer to already be able to detect steganographic images so that it can turn those images into adversarial steganographic images. Doing this during training time is not feasible. Instead, we generate adversarial steganographic images after training and then later update the steganalyzer using these images.

In section 6.3.1, we experiment with how adversarial training affects a steganalyzer's resilience to StegAttack. In section 6.3.2, we experiment with how adversarial training impacts steganalyzer performance on the source mismatch problem and steganographic embedder mismatch problem.

Steganalyzer	P_{MD}
YeNet	97.8%
YeNet-ADV	**63.1%**

Table 6.3: Missed detection probability (P_{MD}) on two steganalyzers using steganographic image test set against StegAttack. YeNet is a normal YeNet steganalyzer and YeNet-ADV is a YeNet steganalyzer updated with adversarial steganographic images.

6.3.1 Defending Against StegAttack

In this experiment, we test whether adversarial training improves a steganalyzer's resilience to StegAttack. We use the following steps:

1. Train a YeNet steganalyzer using a COCO dataset of 9,000x2 training, 1,000x2 validation images embedded with LSB at 0.5 bpp.

2. Apply StegAttack using PGD(40 iterations, $\epsilon = 0.3$) to produce adversarial steganographic images.

3. Using a copy of YeNet, we update the copy with adversarial steganographic images to create YeNet-ADV.

4. Collect a test set of 2,000 steganographic images from COCO with LSB at 0.5 bpp. All images in the test set must meet condition 1 of StegAttack.

5. Apply StegAttack using PGD(40 iterations, $\epsilon = 0.3$) with the test set and report P_{MD}.

Table 6.3 lists the results of the defense experiment. Using these results, we make the following observations:

- **Adversarial training does improve steganalyzer resilience to StegAttack** - We find that adversarial training reduces StegAttack efficacy but not to the point where the steganalyzer can be considered resilient to StegAttack. Still, our results are not conclusive as we only test one attack parameter under a constrained environment. Future experiments will need to further investigate how effective these examples truly are at building steganalyzer resilience to StegAttack.

- **New training methods are needed for better resilience** - In our current setup, we cannot effectively generate adversarial examples during training time. New methods are needed to do this in order to develop better steganalyzer resilience to StegAttack.

6.3.2 Adversarial Training for Universal Steganalysis

Next, we explore how adversarial examples can be used to improve a model's ability for steganographic embedder mismatch problem and the source mismatch problem. In this experiment, we use the following steps:

1. Train two steganalyzers, YeNet and SRNet, using a COCO dataset of 9,000x2 training, 1,000x2 validation images embedded with WOW at 0.5 bpp

2. Apply StegAttack using PGD(40 iterations, $\epsilon = 0.3$) to both YeNet and SRNet to produce two adversarial datasets: SA-YeNet and SA-SRNet.

3. Create two additional steganalyzers by doing the following:

   - **SRNet-MIX** - Update the original SRNet using SA-YeNet.
   - **SRNet-ADV** - Update the original SRNet using SA-SRNet.

4. Test all three steganalyzers (SRNet, SRNet-MIX, SRNet-ADV) on the two following datasets and report P_{MD}:

   - **Change-Embedder Test** - COCO: 5,000x2; HILL at 0.5 bpp
   - **Change-Source Test** - BOSS: 5,000x2; WOW at 0.5 bpp.

Figure 6-4 shows a summary graph of the results from this experiment for SRNet. All raw results can be found in Table B.7. Using these results, we can make the following observations:

- **Adversarial training slightly improves steganographic embedder mismatch problem** - The red line shows steganalyzer performance on the changed

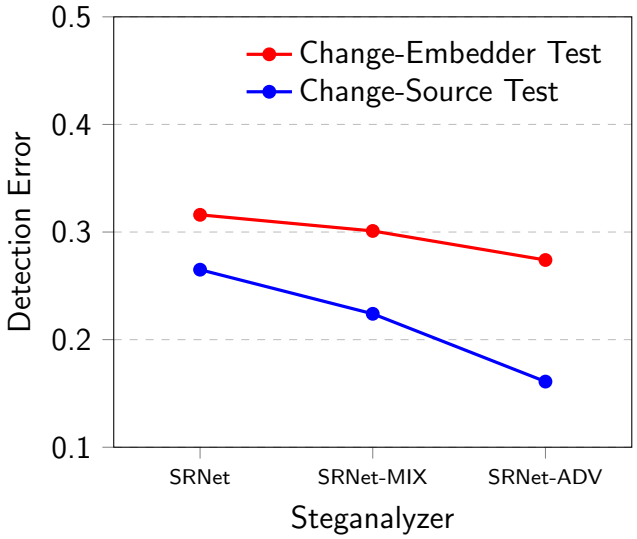

Figure 6-4: The detection error of three different steganalyzers on a changed embedder test set and changed source test set. SRNet is a normal SRNet model, SRNet-MIX is a SRNet model updated with YeNet adversarial steganographic images, and SRNet-ADV is a SRNet model updated with SRNet adversarial steganographic images.

embedder test set. We find that using adversarial training reduces the detection error for steganographic embedder mismatch problem, but not by a very significant amount.

- **Adversarial training does mitigate the source mismatch problem** - The blue line shows steganalyzer performance on the changed source test set. We find that using adversarial training reduces the detection error, which suggests that adversarial examples help prevent over-fitting to source distribution signals. Still, when we compare to reference results in section 5.4, we find that steganalyzer performance is still the best when trained on the BOSS dataset. This makes sense since steganalyzers trained on BOSS can exploit BOSS-specific source distribution signals.

- **Adversarial training using adversarial examples from another steganalyzer provides marginal performance gain** - Across both test sets, we find that training SRNet-MIX on YeNet adversarial steganographic images slightly reduces the detection error. Still, this result may be an effect of using more training samples when training on the adversarial dataset.

- **Adversarial training is a practical method to improve universal steganalysis** - Using adversarial steganographic images from StegAttack is useful for training universal steganalyzers in a practical situation, as they are easy to generate compared to the difficult challenge of generating enough source diversity and steganographic embedder diversity in the training set to properly encompass all possibilities.

Chapter 7

Conclusions and Future Work

In this chapter, we provide concluding remarks and propose several new directions for future research. In section 7.1, we organize our findings into a robust framework for universal and practical steganalysis. In section 7.2, we discuss security strategies that use steganalysis to mitigate steganography-enabled threat models. In section 7.3, we propose exciting areas for future steganalysis research.

7.1 Robust Steganalysis Framework

Guided by the findings discussed in this thesis, we define a three-part framework for universal and practical steganalysis composed of the following techniques:

- **Design Considerations:** Researchers should aim to design a universal steganalyzer whose architecture meets the following requirements: (1) arbitrary image size detection; (2) training and execution efficiency; and (3) robust steganographic signal extraction. The first two requirements allow a steganalyzer to be practically applied while the third requirement allows a steganalyzer to be reliably applied in a real-world situation. In this thesis, we provide promising solutions for each of these requirements. ArbNet shows a promising result for arbitrary image size detection by using a DenseNet architecture along with a spatial pyramid pooling layer. FastNet shows a promising solution for training

and execution efficiency as it is a small network that can perform comparably to state-of-the-art steganalyzers. Finally, steganalyzers like SRNet that use residual-based architectures avoid downsampling steganographic signal, are more performant, and are robust towards adversarial attacks, thereby suggesting that these architectures hold a promising way forward towards meeting the third requirement.

- **Training Procedures:** Researchers should use the following data augmentation techniques to improve their training datasets so that they can train universal steganalyzers: source diversity with an emphasis on textured datasets, steganographic embedder diversity with an emphasis on deep learning-based steganographic embedders, and adversarial steganographic images that can be generated by the StegAttack system. In our experiments, we find that steganalyzer robustness is directly affected by training configurations. Specifically, we show that certain training configurations cause steganalyzers to have issues with source mismatch, steganographic embedder mismatch, and adversarial attacks. Researchers must re-evaluate the training configurations they use so that their steganalyzers can be more universal.

- **Evaluation Mechanisms:** Researchers should leverage StegBench to robustly evaluate steganalyzers in diverse contexts. Robust evaluation of steganalyzers is necessary for the practical deployment of steganalysis. StegBench enables robust steganalysis evaluation through its powerful orchestration and configuration management platform. It supports the comprehensive and practical evaluation of steganalyzers through the use of adversarial attacks and diverse datasets. Using StegBench, we find that current state-of-the-art steganalyzers have a number of failure modes when they are evaluated in diverse contexts. Since most steganalysis research evaluates steganalyzers in limited contexts, their results do not properly capture model robustness. We believe that proper StegBench usage will enable robust steganalysis evaluation.

7.2 Security Controls Using Steganalysis

In Appendix A, we design and demonstrate several steganography-enabled cybersecurity threat models. In general, we find that these threat models have the following in common:

- **Common Transmission Mediums.** Since these threat models deliver malicious information or exploits via steganography, they are by definition making use of the same transmission mediums on which steganography operates - common file formats such as audio, image, and video.
- **Reliance on Proper File Transmission to End Victim.** These threat models depend on the steganographic file reaching the end-victim so that the attack can be completed via the extraction and eventual execution of malicious content. If the steganographic file is not accessible or corrupted, the attack is neutralized.

To defend against steganography-enabled attacks, we investigate how to properly apply steganalysis in two settings: enterprise and personal. In the enterprise setting, we identify critical controls that security systems can use to mitigate steganographic threats. These controls are listed in Table 7.1. In the personal setting, even though there are controls that can mitigate security vulnerabilities, we believe that these controls impose strict usability limits by slowing down computing processes and limiting other functionality. Thus, to mitigate steganalysis on a personal setting, we argue that the best defense consists of network-level steganalysis that checks common transmission medium services for steganographic content (i.e. Twitter[1], Facebook[2]).

In any case, for any effective security mitigation technique to work, the steganalyzers must conform to a robust design, training, and evaluation framework. Furthermore, these models must be continuously updated with adversarial examples to prevent against adversarial attacks. Finally, models must be robust in practical situations (i.e. against different image formats, image resolutions, unknown sources,

[1] `https://twitter.com/`
[2] `https://www.facebook.com/`

Control Mode	Control Process Description
Stricter Firewall	Steganalysis checks on all incoming transmissions to block or flag potentially malicious content
Periodic Checks	As steganalyzers improve, previous undetected transmissions become detectable, and so periodic checks are used to provide adaptive security
Transmission Modification	Flagged malicious content is modified to neutralize potential steganographic content and maintain file usability (i.e. image modification methods).

Table 7.1: List of security controls that employ steganalysis to mitigate steganography-enabled threat models in an enterprise security setting.

and unknown embedders). In the ideal case, a universal steganalyzer should be employed as part of the security infrastructure. However, we believe that ensembles of discriminate steganalyzers can also prove relatively effective. In summary, we reason that by generating robust steganalyzers using our framework along with our proposed security controls, steganography-enabled threat models can be effectively mitigated.

7.3 Future Work

We have identified the following exciting avenues of research in steganalysis:

- **Exploration of Adversarial Attack Methods** - Researching new adversarial attack methods will help identify additional failure modes and help researchers better design and train steganalyzers.
- **Comprehensive Evaluation of Deep Learning Steganographic Embedders** - Deep learning steganographic embedders hold a lot of promise as useful training examples but are also much harder to detect. A comprehensive evaluation will uncover any steganalyzer failure modes that are specific to deep learning steganographic embedders.
- **Design of Steganalysis-Based Mitigation Strategies** - As steganalyzers improve, researchers must design better mitigation strategies that effectively use steganalyzers to neutralize steganography-enabled threat models.

Appendix A

Attacks via Steganography

In this appendix section, we discuss several steganographic attack vectors we have designed and developed. The significant security risks demonstrated by these attack vectors emphasize the need for defense systems that employ robust steganalysis models. We begin by describing steganographic malware systems, which are attack vectors that rely on undetectable transmission. We then showcase proof-of-concept applications to demonstrate how our theorized systems could be deployed in practice. SteganoGAN [47] is used for all demonstrations to show how modern steganography can be integrated into steganography-based attack vectors.

A.1 Malware Systems

In this section, we discuss how steganographic embedders can be applied to malware systems. We identify two types of attack vector that are enabled via steganography: (1) single-pronged attack vector, and (2) steganography-enabled botnets. In the single-pronged attack vector method, we find that steganography aids attackers in hijacking another machine by allowing for the transmission of exploits through undetectable transmission mediums. In botnets, we find that steganography allows for undetectable communication through a variety of transmission mediums, thereby allowing for more powerful control channel communication.

A.1.1 Single-Pronged Attack Vector

The single-pronged attack vector operates by sending exploits to a victim through a steganographic medium. First, as part of the attack setup, a hacker must inject a decoder into the victim's computer, in order to extract the exploit from the steganographic transmission medium. We argue that even though this attack vector depends on such an injection, it is still a powerful and useful attack method, because the decoder is a small piece of benign-looking software. Hackers can use a number of documented methods to do one-time-injection of such software onto a user machine, such as a drive-by download or by including the software as part of an official application.

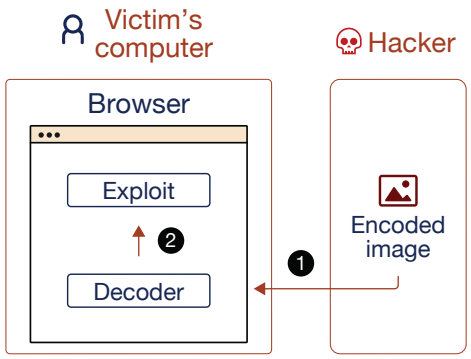

Figure A-1: The single-pronged attack vector uses steganography to deliver undetectable exploits. In the attack setup, a decoder must be preloaded onto the victim machine. Next, during the attack: (1) a hacker transmits an exploit-encoded file (i.e. an image) to the victim's computer and then (2) upon transmission, the decoder loads the file, extracts the exploit, and executes it. The figure shows the browser variant of the attack, in which the decoder is installed on the victim's browser.

Once the decoder is loaded onto the victim machine, the hacker transmits an exploit-encoded file to the victim. The transmission easily bypasses network defenses since it looks like an ordinary file. Upon retrieving the file, the decoder loads the file, extracts the exploit, and executes it. Since the decoder remains on the victim's computer, the hacker can keep sending various exploits through this mechanism to exploit different parts of the victim's machine. Figure A-1 shows an example of this execution process via a browser-based attack. The single-pronged attack vector provides two key features that are enabled by steganography:

1. **Detection-less transmission of exploits**. Steganography uses common and

relatively benign file formats that are only blocked by the strictest of firewalls.

2. **Multiple exploit execution**. Exploit-embedded files can be sent and executed repeatedly until the decoder is neutralized by security defenses.

We demonstrate two applications of the single-pronged attack vector: StegWeb (section A.2), which is a web server-based attack, and StegPlugin (section A.3), which is a browser extension-based attack.

A.1.2 Steganography-Enabled Botnets

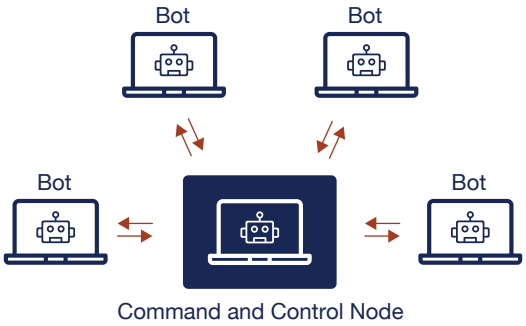

Figure A-2: In this botnet, bots communicate directly with a command and control (CNC) server using control channels to receive and transmit data. Mitigation techniques try to stop the CNC server or control channels. Image Credit: CloudFare[1]

Traditionally, a bot network is a network of computers containing Trojan horses or other malicious code that work together to perform tasks or retrieve compromised user information as specified by the network's controller [28]. Often, bots communicate with a command server via a control channel such as Internet Relay Chat to retrieve tasks and send back user-information. Figure A-2 shows an example botnet architecture where bots communicate with a command and control server.

In recent years, botnets have become more pervasive, prompting security specialists to design various mechanisms to defend against them. One method is to neutralize the control channels that botnets rely on for communication. Neutralizing these channels effectively mitigates the botnet attack vector since bots are unable to communicate information to the control node or receive commands to execute. To

counter this, bot networks are now being designed with more sophisticated communication channels.

We argue that steganography is a powerful and sophisticated communication channel for botnets that poses significant risk. Steganography allows for detection-less transmission of information through a variety of mediums. If botnets are able to leverage steganographic communication channels, it will be very difficult for current security defense mechanisms to block these communication mediums, as any file format could become a transmission medium. We find that steganography-enabled botnets have the following key advantages:

1. **Detection-less transmission of commands and information**. Current bot mitigation techniques will find it difficult to differentiate benign files from steganographic files since they do not employ steganalysis.

2. **Availability of transmission mediums**. Steganography operates on a large variety of file types, giving botnets access to more communication channels.

In section A.4, we present StegCron, a proof-of-concept application that demos a steganography-enabled bot communication channel.

A.2 StegWeb

StegWeb, shown in Figure A-3, is a proof-of-concept system showcasing a variant of the single-pronged attack vector. The system enables a user to hide JavaScript code in normal images, transmit them to a web server, and extract and execute the code on the server.

A.2.1 Attack Description

We now describe the StegWeb attack flow. StegWeb demos a hijacked web service that loads, extracts, and executes arbitrary JavaScript code sent by a bad actor.

[1] `https://www.cloudflare.com/learning/ddos/what-is-a-ddos-botnet/`

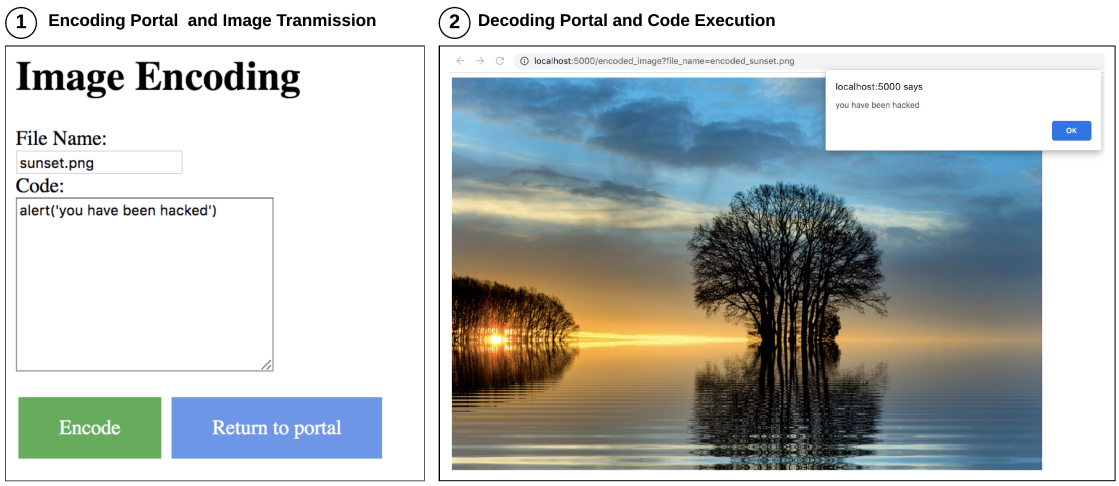

Figure A-3: StegWeb is a proof-of-concept web application that demos the single-pronged attack vector. First, a hacker encodes JavaScript (i.e. the alert message in the figure) on a supplied image. Next, the image is transmitted to the compromised web server, where a decoder extracts the JavaScript and executes it on the web server.

Exploit Transmission

StegWeb provides a web interface to upload and encode an image. Once encoded, the image is transmitted to a file storage system located on the web service. This transmission interface demonstrates how a hacker could use an upload service to transmit the image to the compromised web server. The left-hand side of Figure A-3 shows an example use case where the hacker embeds an alert function inside the user-supplied image.

Exploit Execution

The decoding component extracts the exploit from the image and executes it. In our system, the decoding script periodically checks the image database of the web server and tries to extract messages from any image recently uploaded to the database. If it successfully extracts an exploit from the image, the decoder will then execute this exploit using an evaluate function. We can see the embedded alert message being executed on the right-hand side of Figure A-3. Remember that the decoder is assumed to have been injected into the web server as part of the attack setup.

StegWeb models this assumption by including the decoder in the web server's image handling subroutines.

Next, as long as the decoder remains on the web service, the hacker can repeatedly send exploit-encoded images to hijack and exploit different parts of the web service. Furthermore, steganography prevents modern network defense layers from detecting that an exploit is being transmitted to the decoder. If the hacker was attempting to send a plaintext exploit command to the web server, it is likely that their attempt would be blocked by standard network defense mechanisms.

This attack flow demos how, in practice, a hacker could hijack a web service using steganography. Furthermore, our implementation showcases the potential danger hackers could inflict upon a server if they managed to inject a steganographic decoder.

A.2.2 Software Specification

We developed the demo application using Flask[2] and standard Python libraries for image handling such as DropZone[3]. The code and setup procedures can be found at `https://github.com/DAI-Lab/Stegosploit` under the StegWeb folder.

A.3 StegPlugin

StegPlugin, shown in Figure A-4, is a proof-of-concept system that enables the single-pronged attack vector through a Google Chrome[4] extension. StegPlugin demonstrates how this attack vector could be targeted at common user machines.

A.3.1 Attack Description

We now describe the StegPlugin attack flow. StegPlugin uses Google Chrome extensions to extract, load, and execute arbitrary code from images found during browsing.

[2]https://flask.palletsprojects.com/en/1.1.x/
[3]https://www.dropzonejs.com/
[4]`https://www.google.com/chrome/`

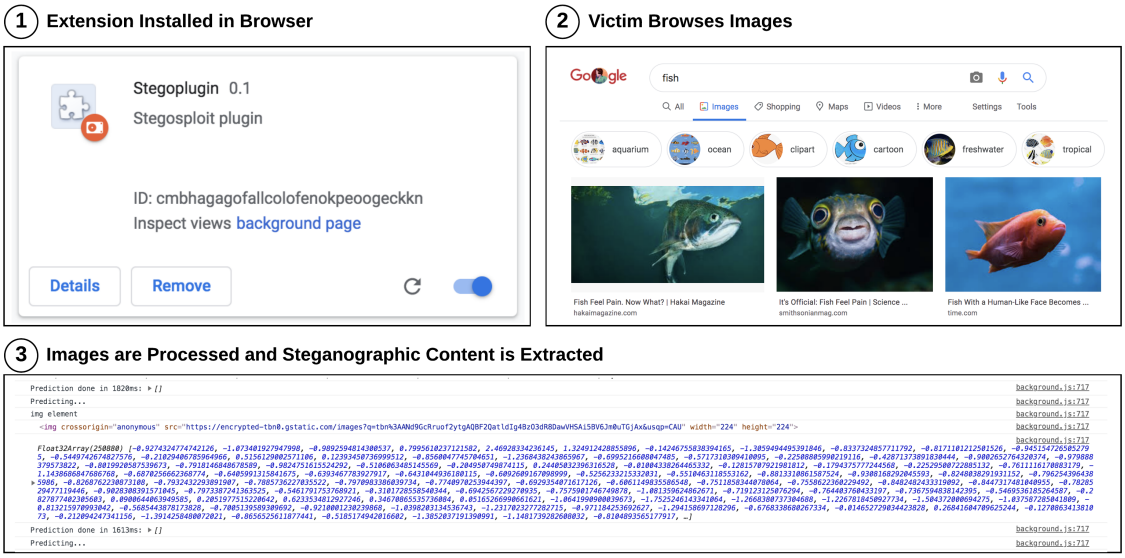

Figure A-4: StegPlugin is a proof-of-concept browser extension that demos the single-pronged attack vector. First, the extension is loaded on the victim machine. Next, the victim browses images (i.e. fish). Finally, StegPlugin fetches all browsed images and attempts to extract and execute any discovered steganographic content.

Extension Installation

As described earlier, the single-pronged attack vector assumes that the decoding script is injected into the victim machine as part of the attack setup. In keeping with this assumption, we assume that the hacker installs the Chrome extension onto the victim machine via standard security exploits such as drive-by downloads.

Message Extraction

In the next part of the attack vector, the machine attempts to extract a message from any image that the user views on their browser. Google Chrome allows extensions to listen for image fetch requests that the browser makes when viewing images. This allows StegPlugin to process all images that the user is looking at. For example, in Figure A-4, the victim browses images of fish on Google Images which leads to a bulk processing of these images by the SteganoGAN decoder. This process models how hackers could easily transmit exploits to victim machines by posting steganographic images on commonly-viewed social media channels.

Arbitrary Code Execution

Once the SteganoGAN decoder extracts an exploit from a browsed image, the extension executes the retrieved exploit, thereby completing the single-pronged attack vector. In our modeling of this attack process, StegPlugin logs the exploit to the console.

This implementation of the single-pronged attack vector shows how steganography enables a powerful exploit-transmission channel for hackers. Since images posted to the web are not checked for steganographic content, hackers can easily transmit this content to the end-victim machine. Furthermore, as soon as the malicious plugin is injected onto the victim browser, hackers can send a multitude of exploits to compromise various parts of the victim machine. Thus, this attack vector further underscores the dangers posed by steganography-enabled attack methods.

A.3.2 Software Specification

To enable SteganoGAN execution in the browser, we use WebGL, a JavaScript API that enables GPU-accelerated execution in the browser. StegPlugin can be found at https://github.com/DAI-Lab/Stegosploit under the StegPlugin folder.

A.4 StegCron

StegCron, shown in Figure A-5, is a proof-of-concept system that showcases steganography-enabled botnet communication channels using cron procedures.

A.4.1 Attack Description

We now describe the StegCron attack flow. StegCron uses steganography-enabled cron tasks to transmit and receive malicious information. StegCron underscores the potential dangers of a steganography-enabled botnet communication channel as current network defenses do not check common file types for steganographic content. Figure A-5 gives a graphic description of the StegCron attack flow.

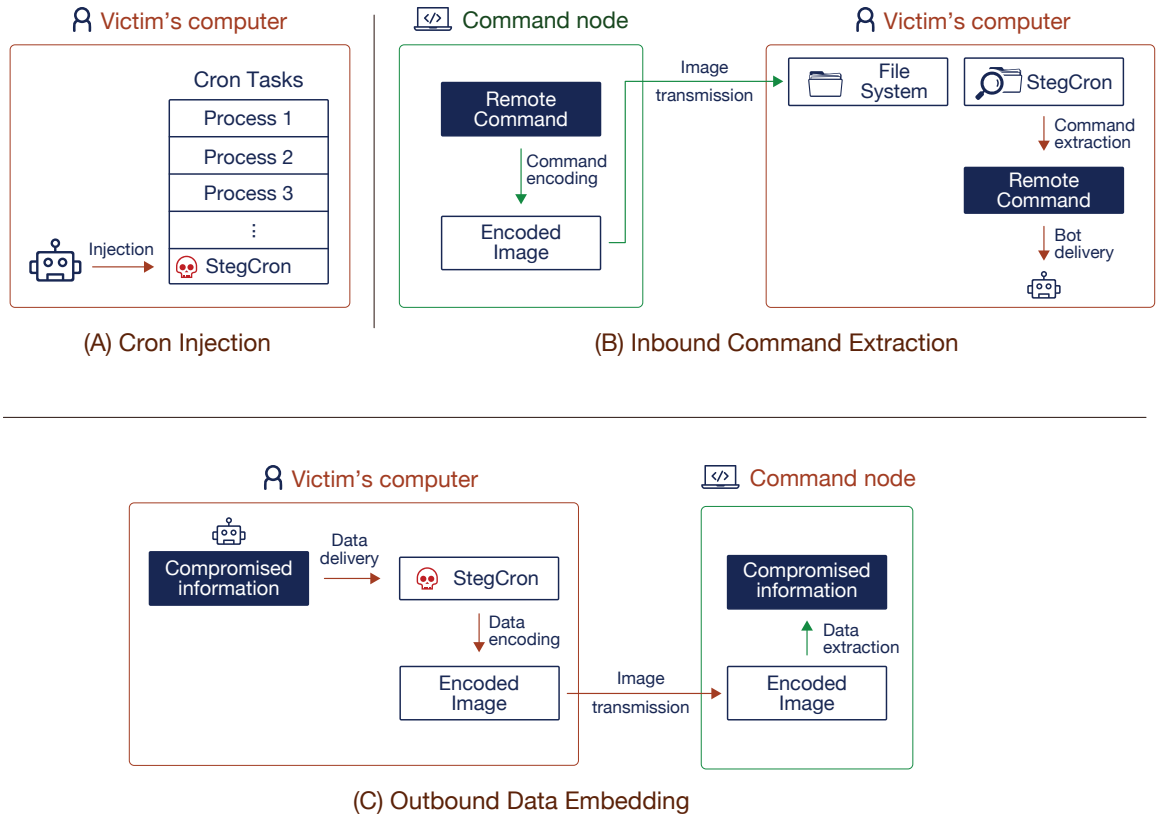

Figure A-5: StegCron is a proof-of-concept cron job system that demos steganography-enabled botnet communication channels. First, the cron system is injected into the victim machine by the bot. Next, the cron system scans any downloaded images for steganographic content, and if found, delivers them to the bot. Finally, the cron system can embed messages and deliver them to the command node.

Cron Injection

Traditionally, when botnets compromise a new machine, the bot comes preprogrammed with procedures on how to communicate with the command node. In our attack setup, we assume that StegCron injection into the user system is part of the bot's preprogrammed hijacking procedure. To model this, we develop an OSX application that when executed injects StegCron into the user system.

Inbound Message Extraction

To receive messages, StegCron uses the following process flow:

1. StegCron listens to a user's file system for new image downloads

2. Newly downloaded images are checked for steganographic content

3. Any extracted steganographic content is delivered to the bot

Because we do not use actual bots, we model step three by having StegCron write the message to a known location. This process flow models how a bot could conceivably extract messages from inbound images. In practice, the communication channel mechanisms would be more sophisticated to ensure the proper delivery of steganographic images. For example, only images with certain attributes would be checked to avoid expensive computational processing and possible detection.

Outbound Message Embedding

To transmit messages, StegCron uses the following process flow:

1. StegCron retrieves messages from the bot on the user machine

2. StegCron embeds messages using images that are already on the user machine

3. StegCron transmits the steganographic image to the command node

Because we do not use actual bots, we model step one by having StegCron periodically embed messages and we model step three by saving the steganographic image to a known location. This process flow models how a bot could transmit messages via steganography. In practice, bots would be careful in choosing the transmission channel and have redundancies in place to ensure message transmission.

A.4.2 Software Specification

StegCron is a set of python scripts that are packaged as an OSX application. The OSX application mimics a bot and injects StegCron using crontab. The code can be found at https://github.com/DAI-Lab/Stegosploit under the StegCron folder.

Appendix B

Results Tables

Experiment Setup		Embedding Ratios				
Embedder	**Detectors**	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**
WOW	YeNet	36.3%	28.2%	23.8%	18.1%	16.2%
	XuNet	36.8%	28.4%	22.4%	16.7%	14.0%
	SRNet	34.4%	**26.1%**	19.2%	**13.7%**	**11.4%**
	ArbNet	**33.9%**	26.8%	**18.4%**	14.2%	11.6%
	FastNet	36.5%	28.1%	22.7%	17.3%	14.4%
S_UNIWARD	YeNet	41.7%	33.2%	27.1%	22.2%	18.9%
	XuNet	41.3%	33.0%	26.3%	20.1%	16.5%
	SRNet	37.5%	29.3%	**21.7%**	**15.5%**	**13.7%**
	ArbNet	**37.1%**	**28.8%**	22.3%	16.1%	14.3%
	FastNet	41.6%	33.7%	26.8%	19.6%	16.3%
HILL	YeNet	43.1%	34.3%	28.7%	24.8%	21.7%
	XuNet	43.3%	33.2%	26.5%	22.3%	18.5%
	SRNet	38.8%	**31.0%**	**23.1**%	19.4%	**15.8**%
	ArbNet	**38.4%**	31.8%	24.0%	**18.8%**	16.1%
	FastNet	43.6%	33.7%	26.8%	22.4%	17.6%

Table B.1: Detection error of five steganalyzers on test sets with various embedders and varying embedding ratios. Detectors are trained with the same configuration as the test set. Bolded metrics correspond to the best performing steganalyzers.

Experiment Setup		Test Image Size (HxW)		
Embedder	**Detectors**	**256x256**	**512x512**	**1024x1024**
WOW	YeNet	16.2%	15.7%	14.8%
	SRNet	**11.4%**	10.9%	10.2%
	ArbNet	12.1%	**10.5**%	**9.4%**
HILL	YeNet	21.7%	21.2%	19.1%
	SRNet	**15.8**%	**15.4%**	14.4%
	ArbNet	16.8%	15.6%	**13.7**%

Table B.2: Detection error of three steganalyzers on test sets with various embedders at 0.5 bpp on three different image resolutions. Detectors are trained with the same configuration as the test set, except for ArbNet which is trained on a mixed-resolution dataset. Bolded metrics correspond to the best performing steganalyzers.

Experiment Setup		Test [BOSS]		Test [COCO]	
Training Embedder	**Training Dataset**	**WOW**	**HILL**	**WOW**	**HILL**
WOW	BOSS	**12.8%**	**33.1%**	17.9%	39.6%
	BOSS+COCO	15.6%	36.5%	12.7%	32.1%
	COCO	27.3%	44.7%	**11.4%**	**31.6%**
HILL	BOSS	**28.8%**	**17.6%**	35.1%	23.5%
	BOSS+COCO	31.7%	20.8%	27.6%	16.6%
	COCO	42.4%	35.3%	**26.9%**	**15.8%**

Table B.3: Detection error of SRNet model trained on steganographic embedder listed in the 'training embedder' column at 0.5 bpp using the dataset listed in the 'training dataset' column and tested against the steganographic embedder listed in the 'test' column using the source specified by the column header. Bolded metrics correspond to the best performing training dataset.

Experiment Setup		Test Embedder			
Training Embedder	**Detector**	**WOW**	**S_UNIWARD**	**HILL**	**S-GAN**
WOW	YeNet	16.2%	22.2%	32.8%	**37.8%**
	SRNet	11.4%	17.7%	31.6%	**36.7%**
S_UNIWARD	YeNet	21.8%	18.9%	30.9%	**38.3%**
	SRNet	16.2%	13.7%	29.3%	**36.5%**
HILL	YeNet	28.2%	30.0%	21.7%	**37.3%**
	SRNet	26.9%	27.5%	15.8%	**35.4%**
S-GAN	YeNet	32.8%	33.6%	**34.3%**	24.6%
	SRNet	29.3%	30.7%	**31.8%**	21.0%

Table B.4: Detection error of two steganalyzers trained on embedders listed in the 'training embedder' column at 0.5 bpp and tested against embedders listed in the 'test embedder' column at 0.5 bpp. Bolded metrics correspond to the hardest embedder to detect.

Experiment Setup		Test Embedder			
Training Embedders	**Detector**	**WOW**	**HILL**	**S-GAN3**	**S-GAN4**
WOW+HILL	YeNet	16.3%	22.7%	36.2%	**36.9%**
	SRNet	12.4%	17.8%	34.2%	**35.1%**
S-GAN1+S-GAN2	YeNet	27.3%	30.7%	**14.2%**	12.4%
	SRNet	26.4%	28.0%	11.1%	**11.7%**

Table B.5: Detection error of two steganalyzers trained on embedders listed in the 'training embedders' column at 0.5 bpp and tested against embedders listed in the 'test embedder' column at 0.5 bpp. Bolded metrics correspond to the hardest embedder to detect.

Training/Test Embedder	COCO	Enhanced
LSB/LSB	2.1%	**50.0%**
WOW+LSB/WOW	16.2%	**31.6%**
HILL+LSB/HILL	21.7%	**35.1%**

Table B.6: Detection error of YeNet model trained on embedder listed in the 'training embedder' column using COCO images and tested against the same embedder at 0.5 bpp using COCO or adversarial images. Bolded metrics correspond to the worst performing source dataset.

Experiment Setup		Test Embedder [BOSS]	
Detector	**Training Dataset**	**WOW**	**HILL**
YeNet	COCO	31.1%	46.1%
	COCO + ENHANCED_YeNet	21.4%	39.9%
	BOSS	**17.3%**	**34.3%**
SRNet	COCO	27.3%	44.7%
	COCO + ENHANCED_YeNet	22.4%	41.1%
	COCO + ENHANCED_SRNet	16.1%	36.8%
	BOSS	**12.8%**	**33.1%**

Table B.7: Detection error of two steganalyzers trained on WOW 0.5 bpp using the dataset listed in the 'training dataset' column and tested against embedders listed in the 'test embedder' column using the BOSS dataset. Bolded metrics correspond to the best performing training dataset.

Appendix C

StegBench

C.1 Configuration Specification

In this section, we give an overview of the configuration file specification. For even more detailed information, please read the instructions provided at http://github.com/DAI-Lab/StegBench.

StegBench works by processing configuration files that contain information about each of the tools in the system. Configurations enable StegBench to be a highly interoperable and modular system that can seamlessly integrate into existing steganographic evaluation pipelines. Configurations specify how a specific steganographic or steganalysis tool operates on a machine and detail the following:

1. **Compatibility**: The tool's compatibility with different operating mechanisms
2. **Command Execution**: Command execution information related to how the tool is executed on the machine

Configuration files are specified by the file format .ini. In any configuration file, there can be a number of configurations for different tools. Each tool is uniquely identified by its header in that file, which is the user-friendly name the system designates to that tool. We list the common configuration options available to each tool in Table C.1.

Arguments	
algorithm_type	specifies whether the tool is an embedder or detector
compatible_types	specifies the image formats the algorithm operates on
command_type	specifies command execution environment
run	specifies the skeleton command for execution
post_run	specifies the skeleton command for post-processing

Table C.1: List of general algorithmic configurations. These configurations specify tool compatibility and execution details and enable tool integration with StegBench.

Arguments	
docker_image	specifies the name of the docker image to us
working_dir	specifies the directory to execute the command from

Table C.2: List of docker specific configurations. These configurations enable StegBench integration with tools dependent on Docker.

Next, StegBench supports two different command execution modes: native and docker. Native commands execute natively while docker commands enable integration with platform-specific tools. Additional docker configurations are listed in Table C.2.

C.1.1 Embedder Configuration

```
[JUNIWARD-COLOR]
algorithm_type = embeddor
compatible_types = ['jpeg', 'jpg']
max_embedding_ratio = 0.5
command_type = docker
docker_image = local/alethia
working_dir = /opt/alethia
run = ./aletheia.py j-uniward-color-sim INPUT_IMAGE_DIRECTORY PAYLOAD OUTPUT_IMAGE_DIRECTORY
output_file = INPUT_IMAGE_NAME
```

Figure C-1: Configuration files are used to specify tool-specific information for embedding algorithms. The figure shows configurations for two embedders. Embedder configurations specify compatible image types, maximum embedding ratios, and skeleton commands for the generation and verification of steganographic datasets.

There are several embedder-specific configuration commands that StegBench allows a user to specify for better compatibility with modern steganography tools. Figure C-1 shows an example embedder configuration file. Table C.3 gives a list of embedder configuration options along with their descriptions. Table C.4 gives a list of embedders that have been shown to integrate with StegBench.

Arguments	
max_embedding_ratio	specifies the embedder's max embedding ratio
verify	specifies a skeleton command for decoding steganographic content to verify proper embedding

Table C.3: List of embedder-specific configuration configurations. These specifications specify embedder compatibility and execution requirements.

Embedder	Mode	Embedder	Mode	Embedder	Mode
F5	Frequency	StegHide	Frequency	SteganoLSB	Spatial
J_UNIWARD	Frequency	WOW	Spatial	LSBR	Spatial
JSteg	Frequency	HUGO	Spatial	CloackedPixel	Spatial
Outguess	Frequency	S_UNIWARD	Spatial	OpenStego	Spatial
UED	Frequency	MiPOD	Spatial	HiDDeN	DL
EBS	Frequency	HILL	Spatial	SteganoGAN	DL

Table C.4: List of 18 embedders that have successfully worked with StegBench

C.1.2 Detector Configuration

There are several detector-specific configuration commands that StegBench allows a user to specify to allow for better compatibility with modern steganalysis tools. Figure C-2 shows an example detector configuration file. Table C.5 gives a list of detector configuration options along with their descriptions. Table C.6 gives a list of embedders that have been shown to integrate with StegBench.

Arguments	
detector_type	specifies if a detector is binary or probabilistic
regex_filter_yes	specifies detector output on a steganographic image
regex_filter_no	specifies detector output on a cover image

Table C.5: List of detector-specific configurations. These specifications specify detector compatibility and execution modes.

Detector	Mode	Detector	Mode	Detector	Mode
RS	Stat	StegExpose	Stat	Yedroudj-Net	DL
SPA	Stat	Weighted-Stego	Stat	SRNet	DL
SVM Methods	Stat	Calibration	Stat	XuNet	DL
EC + RM	Stat	QNet	DL	YeNet	DL

Table C.6: List of 12 detectors that have successfully worked with StegBench

```
[spa]
algorithm_type = detector
detector_type = binary
compatible_types = ['png', 'pgm']
command_type = docker
docker_image = local/alethia
working_dir = /opt/alethia/
run = ./aletheia.py spa INPUT_IMAGE_PATH
pipe_output = True
regex_filter_yes = Hiden data found
regex_filter_no = No hidden data found
```

Figure C-2: Configuration files are used to specify tool-specific information for detection algorithms. The figure shows configurations for two detectors. Detector configurations specify compatible image types, skeleton commands for steganalysis, and any result processing-specific requirements.

C.1.3 Command Generation

In this section, we detail how StegBench processes command skeletons to generate executable commands for steganographic and steganalysis processing. StegBench provides a set of flags that are used to define skeleton commands. Skeleton commands are then processed, and flags are substituted appropriately during run-time. These flags can be categorized into three categories: (1) input flags, (2) embedding flags, and (3) output flags. Table C.7 lists each of these flags along with their accompanying descriptions.

C.2 API Specification

In this next section, we detail the API specification of StegBench.

Input Flags	
INPUT_IMAGE_PATH	Substituted for the path of the image that the embedding/detection operation is being applied to
Embedding Flags	
SECRET_TXT	Substituted for the message to be embedded
PASSWORD	Substituted with a password, which is most commonly used to scramble the secret text
PAYLOAD	Substituted with the embedding ratio required during the embedding operation
Output Flags	
OUTPUT_IMAGE_PATH	Substituted for the path to the output image
RESULT_FILE	Substituted for the path to a result file where the detection result will be stored

Table C.7: List of flags used in skeleton commands as part of tool configuration. StegBench uses its orchestration engine and command generation protocols to substitute flags in skeleton commands with appropriate command parameters.

initialize()	
Purpose	
Initializes the StegBench file system in the current working directory	
Output	
None	
add_config(config=[], directory=[])	
Purpose	
Loads and processes tool configurations into the StegBench file system	
Arguments	
config	list of paths to configuration files
directory	list of paths to directories that contain config files
Output	
None	
info(all=True, dataset=True, embedder=True, detector=True)	
Purpose	
Provides StegBench system information including asset UUIDs	
Arguments	
all	if true, shows all available StegBench asset information
dataset	if true, shows all dataset asset information
embedder	if true, shows all embedder asset information
detector	if true, shows all detector asset information
Output	
Requested system information including UUIDs of all requested assets	

Table C.8: StegBench API for system initialization and integration.

add_embedder(embedder_uuids, set_uuid=None)	
Purpose	
Adds to or creates a new embedder set, which is a collection of embedders	
Arguments	
embedder_uuids	a list of UUIDs, each of which specifies an embedder
set_uuid	UUID that specifies an existing embedder set
Output	
UUID of the generated embedder set	
add_detector(detector_uuids, set_uuid=None)	
Purpose	
Adds to a or creates a new detector set, which is a collection of detectors	
Arguments	
detector_uuids	a list of UUIDs, each of which specifies a detector
set_uuid	UUID that specifies an existing detector set.
Output	
UUID of the generated detector set	

Table C.9: StegBench API for algorithmic set generation processes.

download(routine, name, operation_dict={}, metadata_dict={})	
Purpose	
Downloads a specified dataset and applies image/dataset operations to it	
Arguments	
routine	specifies a pre-loaded download routine [ALASKA|BOSS|BOWS2|COCO|DIV2K]
name	specifies a user-friendly name for the dataset
operation_dict	applies specified operation(s) to each image [noise | resize | rotate | center_crop | rgb2gray]
metadata_dict	applies operation(s) on the dataset-level [ttv_split|limit]
Output	
UUID of the downloaded dataset	
process(directory, name, operation_dict={}, metadata_dict={})	
Purpose	
Processes a local dataset and applies image/dataset operations to it	
Arguments	
directory	specifies the path to the dataset to process
name	specifies a user-friendly name for the dataset
operation_dict	applies specified operation(s) to each image [noise | resize | rotate | center_crop | rgb2gray]
metadata_dict	applies operation(s) on the dataset-level [ttv_split|limit]
Output	
UUID of the processed dataset	

Table C.10: StegBench API for the dataset pipeline.

embed(embedder_set_uuid, dataset_uuid, ratio, name)	
Purpose	
Embeds a cover dataset with a set of embedders at a specified embedding ratio	
Arguments	
embedder_set_uuid	UUID of the embedder set being used
dataset_uuid	UUID of the cover dataset being used
ratio	specifies an embedding ratio
name	specifies a user-friendly name for the steganographic db
Output	
UUID of the generated steganographic dataset	
verify(stego_db_uuid)	
Purpose	
Verifies a steganographic dataset	
Arguments	
stego_db_uuid	UUID of the steganographic db to verify
Output	
Results detailing if steganographic images were embedded correctly	

Table C.11: StegBench API for the embedding pipeline.

detect(detector_set_uuid, dataset_uuids)	
Purpose	
Analyzes a set of detectors on a set of datasets.	
Arguments	
detector_set_uuid	UUID of the detector set being used
dataset_uuids	list of UUIDs of cover or steganographic datasets to be detected
Output	
Summary statistics that detail steganalysis results over the supplied datasets	
adv_attack(model_path, dataset_uuids, configurations)	
Purpose	
Applies StegAttack to generate adversarial images against a steganalysis model	
Arguments	
model_path	path to the deep-learning steganalysis model, only PyTorch is currently supported
dataset_uuids	list of UUIDs of steganographic dataset to use as source images for the attack
configurations	specifies attack system configurations including which robustness attack to use [FGSM | EAD | BIM | PGD | UPD] and other attack-specific parameters.
Output	
UUID of adversarially-generated image dataset	

Table C.12: StegBench API for the detection pipeline.

Bibliography

[1] Anish Athalye, Logan Engstrom, Andrew Ilyas, and Kevin Kwok. Synthesizing robust adversarial examples, 2017.

[2] Shumeet Baluja. Hiding images in plain sight: Deep steganography. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 2069–2079. Curran Associates, Inc., 2017.

[3] Patrick Bas, Tomáš Filler, and Tomáš Pevný. "break our steganographic system": The ins and outs of organizing boss. In Tomáš Filler, Tomáš Pevný, Scott Craver, and Andrew Ker, editors, *Information Hiding*, pages 59–70, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[4] M. Boroumand, M. Chen, and J. Fridrich. Deep residual network for steganalysis of digital images. *IEEE Transactions on Information Forensics and Security*, 14(5):1181–1193, May 2019.

[5] M. Boroumand, M. Chen, and J. Fridrich. Deep residual network for steganalysis of digital images. *IEEE Transactions on Information Forensics and Security*, 14(5):1181–1193, 2019.

[6] Marc Chaumont. Deep learning in steganography and steganalysis from 2015 to 2018. *ArXiv*, abs/1904.01444, 2019.

[7] Logan Engstrom, Andrew Ilyas, and Anish Athalye. Evaluating and understanding the robustness of adversarial logit pairing, 2018.

[8] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Brandon Tran, and Aleksander Madry. Learning perceptually-aligned representations via adversarial robustness. *ArXiv*, abs/1906.00945, 2019.

[9] F. Farhat and S. Ghaemmaghami. Towards blind detection of low-rate spatial embedding in image steganalysis. *IET Image Processing*, 9(1):31–42, 2015.

[10] Clément Fuji Tsang and Jessica Fridrich. Steganalyzing images of arbitrary size with cnns. volume 2018, 01 2018.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[12] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

[13] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples, 2014.

[14] L. Guo, J. Ni, and Y. Q. Shi. An efficient jpeg steganographic scheme using uniform embedding. In *2012 IEEE International Workshop on Information Forensics and Security (WIFS)*, pages 169–174, 2012.

[15] Vojtech Holub and Jessica J. Fridrich. Designing steganographic distortion using directional filters. *2012 IEEE International Workshop on Information Forensics and Security (WIFS)*, pages 234–239, 2012.

[16] Vojtech Holub, Jessica J. Fridrich, and Tomás Denemark. Universal distortion function for steganography in an arbitrary domain. *EURASIP Journal on Information Security*, 2014:1–13, 2014.

[17] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269, 2017.

[18] Andrew D. Ker, Patrick Bas, Rainer Böhme, Rémi Cogranne, Scott Craver, Tomás Filler, Jessica J. Fridrich, and Tomás Pevný. Moving steganography and steganalysis from the laboratory into the real world. In *IHMMSec '13*, 2013.

[19] Alexey Kurakin, Ian J. Goodfellow, Samy Bengio, Yinpeng Dong, Fangzhou Liao, Ming Liang, Tianyu Pang, Jun Zhu, Xiaolin Hu, Cihang Xie, Jianyu Wang, Zhishuai Zhang, Zhou Ren, Alan L. Yuille, Sangxia Huang, Yao Zhao, Yuzhe Zhao, Zhonglin Han, Junjiajia Long, Yerkebulan Berdibekov, Takuya Akiba, Seiya Tokui, and Motoki Abe. Adversarial attacks and defences competition. *ArXiv*, abs/1804.00097, 2018.

[20] Bin Li, Junhui He, Jiwu Huang, and Y.Q. Shi. A survey on image steganography and steganalysis. *Journal of Information Hiding and Multimedia Signal Processing*, 2, 05 2011.

[21] Bin Li, Ming Wang, Jiwu Huang, and Xiaolong Li. A new cost function for spatial image steganography. *2014 IEEE International Conference on Image Processing (ICIP)*, pages 4206–4210, 2014.

[22] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft coco: Common objects in context. *ArXiv*, abs/1405.0312, 2014.

[23] Ivans Lubenko. Towards robust steganalysis: binary classifiers and large, heterogeneous data. 2013.

[24] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks, 2017.

[25] W. Mazurczyk and L. Caviglione. Information hiding as a challenge for malware detection. *IEEE Security Privacy*, 13(2):89–93, 2015.

[26] R. Mishra and P. Bhanodiya. A review on steganography and cryptography. In *2015 International Conference on Advances in Computer Engineering and Applications*, pages 119–122, 2015.

[27] Jennifer Newman, Li Lin, Wenhao Chen, Stephanie Reinders, Yangxiao Wang, Min Wu, and Yong Guan. Stegoappdb: a steganography apps forensics image database, 2019.

[28] Norton. Bot network. `http://us.norton.com/online-threats/glossary/b/bot-network.html.`, 2020.

[29] Tomáš Pevný, Tomáš Filler, and Patrick Bas. Using high-dimensional image models to perform highly undetectable steganography. In Rainer Böhme, Philip W. L. Fong, and Reihaneh Safavi-Naini, editors, *Information Hiding*, pages 161–177, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[30] N. Prokhozhev, O. Mikhailichenko, A. Sivachev, D. Bashmakov, and A. Korobeynikov. Passive steganalysis evaluation: Reliabilities of modern quantitative steganalysis algorithms. In Ajith Abraham, Sergey Kovalev, Valery Tarassov, and Václav Snášel, editors, *Proceedings of the First International Scientific Conference "Intelligent Information Technologies for Industry" (IITI'16)*, pages 89–94, Cham, 2016. Springer International Publishing.

[31] Yinlong Qian, Jing Dong, Wei Wang, and Tieniu Tan. Deep learning for steganalysis via convolutional neural networks. In Adnan M. Alattar, Nasir D. Memon, and Chad D. Heitzenrater, editors, *Media Watermarking, Security, and Forensics 2015*, volume 9409, pages 171 – 180. International Society for Optics and Photonics, SPIE, 2015.

[32] Uri Shaham, Yutaro Yamada, and Sahand N. Negahban. Understanding adversarial training: Increasing local stability of supervised models through robust optimization. *Neurocomputing*, 307:195–204, 2018.

[33] Brijesh Singh, Prasen Kumar Sharma, Rupal Saxena, Arijit Sur, and Pinaki Mitra. A new steganalysis method using densely connected convnets. In Bhabesh Deka, Pradipta Maji, Sushmita Mitra, Dhruba Kumar Bhattacharyya, Prabin Kumar Bora, and Sankar Kumar Pal, editors, *Pattern Recognition and Machine Intelligence*, pages 277–285, Cham, 2019. Springer International Publishing.

[34] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*, 2019.

[35] P. Thiyagarajan, G. Aghila, and V. Prasanna Venkatesan. Stego-image generator (SIG) - building steganography image database. *CoRR*, abs/1206.2586, 2012.

[36] Chang Wang and Jiangqun Ni. An efficient jpeg steganographic scheme based on the block entropy of dct coefficients. pages 1785–1788, 03 2012.

[37] Pengfei Wang, Zhihui Wei, and Liang Xiao. Spatial rich model steganalysis feature normalization on random feature-subsets. *Soft Computing*, 12 2016.

[38] Andreas Westfeld. F5—a steganographic algorithm. In Ira S. Moskowitz, editor, *Information Hiding*, pages 289–302, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[39] Stephen B. Wicker. *Reed-Solomon Codes and Their Applications*. IEEE Press, 1994.

[40] S. Wu, S. Zhong, and Y. Liu. A novel convolutional neural network for image steganalysis with shared normalization. *IEEE Transactions on Multimedia*, 22(1):256–270, 2020.

[41] Shengli Wu, Sheng hua Zhong, Yan Liu, and Mengyuan Liu. Cis-net: A novel cnn model for spatial image steganalysis via cover image suppression. *ArXiv*, abs/1912.06540, 2019.

[42] Guanshuo Xu. Deep convolutional neural network to detect j-uniward. In *IH-MMSec '17*, 2017.

[43] Zhongliang Yang, Ke Wang, Sai Ma, Yongfeng Huang, Xiangui Kang, and Xianfeng Zhao. Istego100k: Large-scale image steganalysis dataset. In *IWDW*, 2019.

[44] J. Ye, J. Ni, and Y. Yi. Deep learning hierarchical representations for image steganalysis. *IEEE Transactions on Information Forensics and Security*, 12(11):2545–2557, Nov 2017.

[45] Mehdi Yedroudj, Frédéric Comby, and Marc Chaumont. Yedroudj-net: An efficient cnn for spatial steganalysis. *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2092–2096, 2018.

[46] J. Zeng, S. Tan, B. Li, and J. Huang. Large-scale jpeg image steganalysis using hybrid deep-learning framework. *IEEE Transactions on Information Forensics and Security*, 13(5):1200–1214, 2018.

[47] Kevin Alex Zhang, Alfredo Cuesta-Infante, Lei Xu, and Kalyan Veeramachaneni. Steganogan: High capacity image steganography with gans, 2019.

[48] Xinpeng Zhang, Shuozhong Wang, and Kaiwen Zhang. Steganography with least histogram abnormality. pages 395–406, 09 2003.

[49] Xunpeng Zhang, Xiangwei Kong, Pengda Wang, and Bo Wang. Cover-source mismatch in deep spatial steganalysis. In Hongxia Wang, Xianfeng Zhao, Yunqing Shi, Hyoung Joong Kim, and Alessandro Piva, editors, *Digital Forensics and Watermarking*, pages 71–83, Cham, 2020. Springer International Publishing.

[50] Yiwei Zhang, Weiming Zhang, Kejiang Chen, Jiayang Liu, Yujia Liu, and Nenghai Yu. Adversarial examples against deep neural network based steganalysis. pages 67–72, 06 2018.

[51] Stephan Zheng, Yang Song, Thomas Leung, and Ian J. Goodfellow. Improving the robustness of deep neural networks via stability training. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4480–4488, 2016.

[52] Jiren Zhu, Russell Kaplan, Johanna E. Johnson, and Li Fei-Fei. Hidden: Hiding data with deep networks. *ArXiv*, abs/1807.09937, 2018.