Dipartimento di Matematica e Fisica
Master's Thesis in Computational Science

Contrastive Learning for Linux IoT Malware Classification

Candidate:
Claudio Maione

Supervisor:
Marco Carli
Co-Supervisor:
Francesco Calabrò

Academical Year 2022/2023

Ave atque vale.

Catullus, "*Carme* CI"

Abstract

The main purpose of this thesis is the study and application of computer vision techniques based on Artificial Intelligence (AI) to malware classification.
We approach this topic by first replicating the results obtained by Yueming Wu *et al.* in their paper regarding Android malware classification [WDZ+22]. Afterwards, we extend those ideas to the Linux IoT environment.
In the beginning, we present the basic theory and techniques of Deep Learning, with special focus on "Residual Neural Networks" (ResNets) architectures and "Contrastive Learning" training approach.
Next, we explain the concept of the Internet of Things (IoT) and its cybersecurity challenges, with a particular focus on static malware analysis using Radare2.
Finally, we begin showing the development of the Android and Linux IoT malware classifiers, which are almost entirely performed, via Secure Shell (SSH), on a remote Linux server using the Docker platform. The codebase is written in Python using specialized libraries and frameworks for AI model development such as PyTorch and CUDA.
Lastly, we show some ideas for improvements and future works.

Contents

List of Figures

List of Tables

Chapter 1

Preliminaries

1.1 General Definitions

1.1.1 Kullback-Leibler Divergence

The relative entropy or Kullback-Leibler divergence between two probability distributions defined on the same sample space $\mathcal{X}$ is defined [Mac03] as:

$$D_{KL}(P\|Q) = \sum_{x\in\mathcal{X}} P(x)\log\left(\frac{P(x)}{Q(x)}\right) \tag{1.1}$$

For distributions P and Q of a continuous random variable, relative entropy is defined [Bis06] to be the integral:

$$D_{KL}(P\|Q) = \int_{-\infty}^{+\infty} p(z)\log\left(\frac{p(z)}{q(z)}\right)dz \tag{1.2}$$

where p and q denote the probability densities of P and Q.

1.1.2 Dirac Delta Function

Paul Dirac, in an effort to create the mathematical tools for the development of quantum field theory [Dir81], introduced the Dirac delta function (δ-function), a quantity depending on a parameter x satisfying the conditions

$$\begin{aligned} &\int_{-\infty}^{+\infty}\delta(x)dx = 1 \\ &\delta(x) = 0 \quad \text{for} \quad x \neq 0 \end{aligned} \tag{1.3}$$

$\delta(x)$ is not a function of x according to the usual mathematical definition of a function, which requires a function to have a definite value for each point in its domain, but is something more general, which is called an "improper function".
The most important property of $\delta(x)$ is that

$$\int_{-\infty}^{+\infty} f(x)\delta(x-a)dx = f(a) \tag{1.4}$$

with $a \in \mathbb{R}$.

1.1.3 Softmax

The softmax function converts a vector of K real numbers into a probability distribution of K possible outcomes.

$$\begin{aligned} \sigma : \mathbb{R}^K &\to \left\{ z \in \mathbb{R}^K \mid z_i > 0, \sum_{i=1}^{K} z_{i=1} \right\} \\ \sigma(z)_j &= \frac{e^{z_j}}{\sum_{i=1}^{K} e^{z_i}} \quad \text{for} \quad j = 1 \dots K \end{aligned} \tag{1.5}$$

It is often used as the last activation function of a neural network to normalize the output of a network to a probability distribution over predicted output classes.

1.1.4 Normal Distribution

Given two parameters $\mu, \sigma \in \mathbb{R}$, with $\sigma > 0$, a normal distribution, or Gaussian distribution, is a type of continuous probability distribution for a real-valued random variable.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{1.6}$$

μ is the mean, or expectation, of the distribution, while the parameter σ is its standard deviation and σ^2 is the variance.

1.2 Machine Learning

Machine learning is a subfield of artificial intelligence (AI) that focuses on the development of algorithms and models that enable computers to learn and make predictions or decisions without being explicitly programmed. The fundamental idea behind machine learning is to give computers the ability to automatically learn from data and improve their performance over time.
In traditional programming, developers write explicit instructions for a computer to perform a task. In contrast, machine learning systems learn from examples or data to generalize patterns and make predictions or decisions without being explicitly programmed for a specific task.

1.3 Internet of Things

The Internet of Things (IoT) refers to a network of interconnected physical devices that communicate and exchange data with each other over the internet. These devices are embedded with sensors, software, and other technologies that enable them to collect and exchange data. The goal of IoT is to create a smart, interconnected system where devices can interact and make intelligent decisions without human intervention.
IoT devices can be found in various industries and everyday objects, such as smart home devices, industrial machinery, wearable fitness trackers, healthcare equipment, and more. The data generated by these devices can be collected, analyzed, and used to improve efficiency, make informed decisions, and enhance user experiences.

1.4 MD5, SHA-1 and SHA-256

Message-Digest Algorithm 5 (MD5), Secure Hash Algorithm 1 (SHA-1) and Secure Hash Algorithm 256 (SHA-256) are cryptographic hash functions, *i.e.* non-invertible functions that map a string of arbitrary length to a string of predefined length. They are required to have the following properties:

- **Preimage resistance**: It should be computationally infeasible to find an input string that produces a hash equal to a given hash.
- **Second preimage resistance**: It should be computationally infeasible to find an input string that produces a hash equal to that of a given string.
- **Collision resistance**: It should be computationally infeasible to find a pair of input strings that produce the same hash.

These hash functions are commonly used in various security applications and protocols to ensure data integrity and create digital signatures.

1. **Message-Digest Algorithm 5**: MD5 produces a 128-bit (16-byte) hash value, typically expressed as a 32-character hexadecimal number.
   Widely used in the past for integrity checking and checksums, but it is now considered weak in terms of collision resistance, and its use in security-sensitive applications is not recommended.
2. **Secure Hash Algorithm 1**: SHA-1 produces a 160-bit (20-byte) hash value, typically expressed as a 40-character hexadecimal number.
   Like MD5, SHA-1 has been found to have vulnerabilities to collision attacks, where two different inputs can produce the same hash value. Due to these vulnerabilities, SHA-1 is no longer considered secure for cryptographic purposes.
3. **Secure Hash Algorithm 256**: SHA-256 produces a 256-bit (32-byte) hash value, typically expressed as a 64-character hexadecimal number.
   It is widely used and considered secure for cryptographic purposes. It is commonly used in digital signatures, certificate generation, and various security protocols.

Chapter 2

Deep Learning

Deep Neural Networks are a type of machine learning model that are considered, as of the time of writing, the most powerful and practical machine learning models, with applications in computer vision, speech recognition, natural language processing, and many more.
Machine learning models can be coarsely be divided into three areas: supervised, unsupervised, and reinforcement learning. In this thesis we will focus solely on the Supervised Deep Learning.
To be able to properly explain the importance and power of Deep Neural Networks, we will utilize as reference the book "Understanding Deep Learning" by Simon J.D. Prince [Pri24].

2.1 Supervised Learning

In supervised learning, we aim to build a model that takes an input x and outputs a prediction y, while using a labeled dataset, where the input data is paired with corresponding output labels. To make the prediction, we need a model $f[\bullet]$ that takes input x and returns y. The model is just a mathematical equation with a fixed form that contains parameters ϕ, which determine the particular relation between input and output. When we talk about learning or training a model, we mean that we attempt to find parameters that make sensible output predictions from the input. We learn these parameters using a training dataset of I pairs of input and output examples $\{x_i, y_i\}$. We aim to select parameters that map each training input to its associated output as closely as possible. We quantify the degree of mismatch in this mapping with the loss L, a scalar value that summarizes how poorly the model predicts the training outputs from their corresponding inputs for parameters ϕ. When we train the model, we are seeking parameters $\hat{\phi}$ that minimize this loss function:

$$\hat{\phi} = \underset{\phi}{\operatorname{argmin}}\Big[L[\{x_i, y_i\}, \phi]\Big] \tag{2.1}$$

After training a model, we run the model on separate test data to see how well it generalizes to examples that it didn't observe during training. If the performance is adequate, then we are ready to deploy the model.

2.2 Shallow Neural Networks

Shallow neural networks are functions $y = f[x, \phi]$ with parameters ϕ that map a multi-dimensional input $x \in \mathbb{R}^{D_i}$ to a multi-dimensional output $y \in \mathbb{R}^{D_o}$ using $h \in \mathbb{R}^{D}$ hidden

units. Each hidden unit is computed as:

$$h_d = a\left[\theta_{d0} + \sum_{i=1}^{D_i} \theta_{di} x_i\right] \tag{2.2}$$

and these are combined linearly to create the output:

$$y_j = \phi_{j0} + \sum_{d=1}^{D} \phi_{jd} h_j \tag{2.3}$$

where $a[\bullet]$ is a nonlinear activation function. The activation function permits the model to describe nonlinear relations between input and the output, and as such, it must be nonlinear itself; with no activation function, or a linear activation function, the overall mapping from input to output would be restricted to be linear. Figure 2.1 shows an example with three inputs, three hidden units, and two outputs.

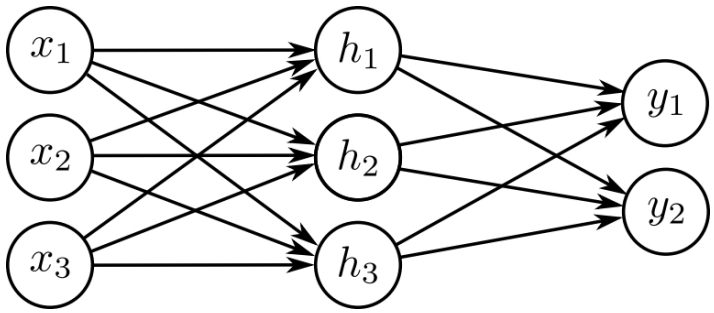


Figure 2.1: Visualization of neural network with three inputs and two outputs.

Many different activation functions exist in the literature; Figure 2.3 illustrates the most common ones, which we will not get into much detail in this thesis. The most common one is the Rectified linear unit (ReLU, Figure 2.2), which has the merit of being easily interpretable:

$$\text{ReLU}\,[z] = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases} \tag{2.4}$$

With ReLU activations, the network divides the input space into convex polytopes where all points in the same linear region activate the same nodes. Each convex polytope contains a different linear function. The polytopes are the same for each output node, but the linear functions they contain can differ. For more information about these linear regions, consult [ZW20].

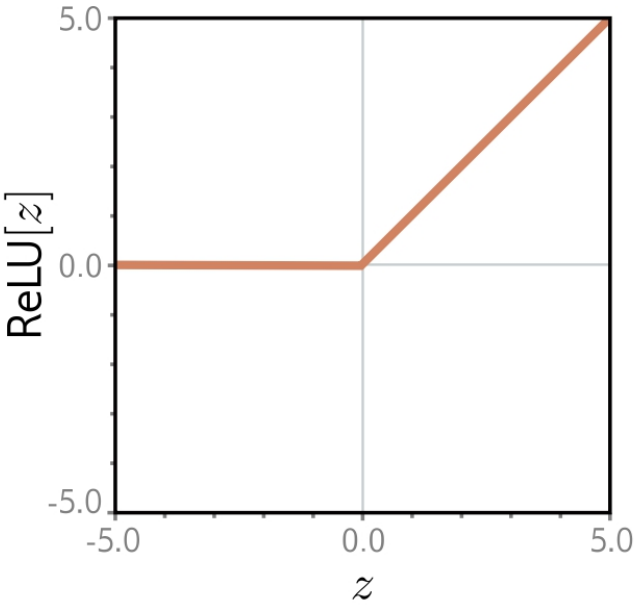


Figure 2.2: Rectified linear unit (ReLU).

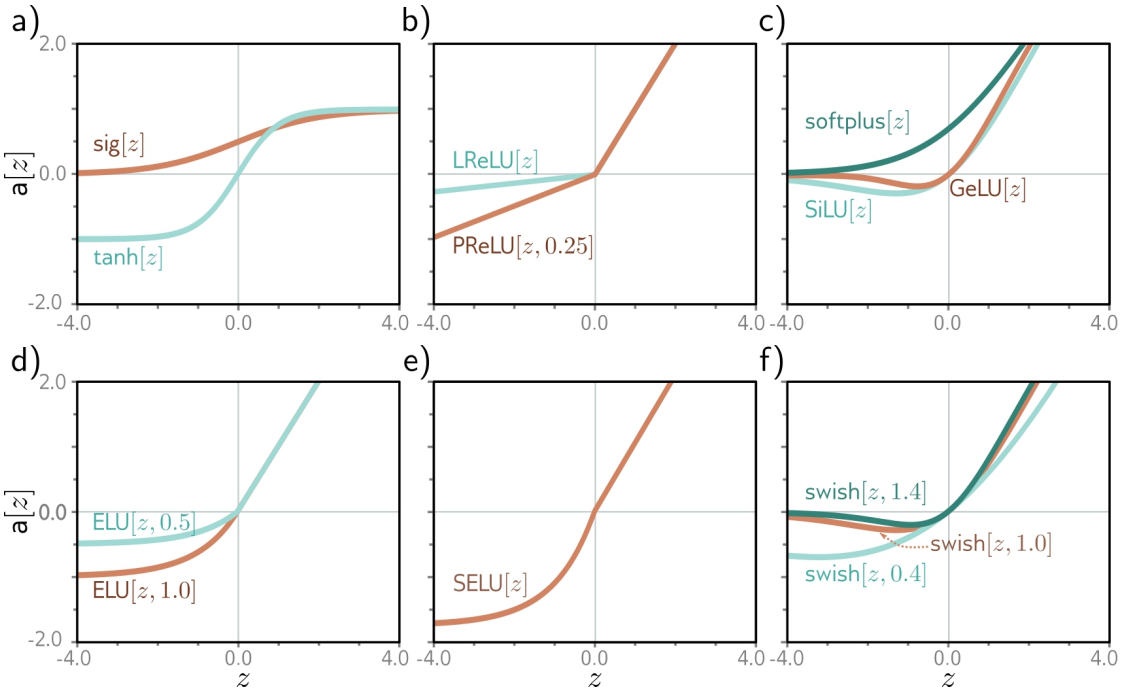


Figure 2.3: Activation functions. a) Logistic sigmoid and tanh functions. b) Leaky ReLU and parametric ReLU with parameter 0.25. c) SoftPlus, Gaussian error linear unit, and sigmoid linear unit. d) Exponential linear unit with parameters 0.5 and 1.0, e) Scaled exponential linear unit. f) Swish with parameters 0.4, 1.0, and 1.4.

2.3 Deep Neural Networks

As the number of hidden units increases, shallow neural networks improve their descriptive power. Indeed, with enough hidden units, shallow networks can describe arbitrarily complex functions in high dimensions. However, it turns out that for some functions, the required number of hidden units is impractically large. Deep networks can produce many more linear regions than shallow networks for a given number of parameters. Hence, from a practical standpoint, they can be used to describe a broader family of functions. Modern networks might have more than a hundred layers with thousands of hidden units at each layer. The number of hidden units in each layer is referred to as the width of the network, and the number of hidden layers as the depth. The total number of hidden units is a measure of the network's capacity.

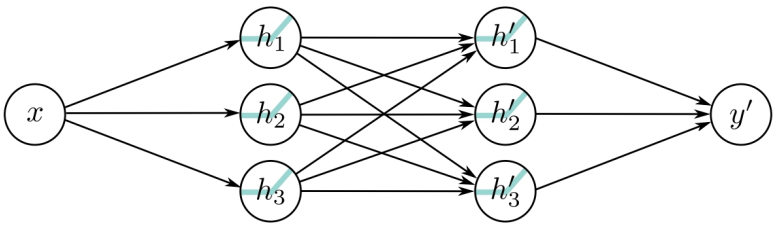


Figure 2.4: Neural network with one input, one output, and two hidden layers, each containing three hidden units.

In the general case of a deep network with input $x \in \mathbb{R}^{D_0}$, output $y \in \mathbb{R}^{D_{out}}$, depth n and $h_k \in \mathbb{R}^{D_k}$ hidden units at depth k, the hidden layers are defined by:

$$h_{k,d} = a\left[\theta_{k,d0} + \sum_{i=1}^{D_{k-1}} \theta_{k,di} h_{k-1,d}\right] \tag{2.5}$$

while the output is computed as follows:

$$y_j = \phi_{j0} + \sum_{d=1}^{D_{n-1}} \phi_{jd} h_{n-1,j} \tag{2.6}$$

Both deep and shallow networks can model arbitrary functions, but some functions can be approximated much more efficiently with deep networks. Functions have been identified that require a shallow network with exponentially more hidden units to achieve an equivalent approximation to that of a deep network. This phenomenon is referred to as the depth efficiency of neural networks.

2.4 Loss Functions

A loss function or cost function $L[\{x_i, y_i\}, \phi]$ returns a single number that describes the mismatch between the model predictions $f[x_i, \phi]$ and their corresponding ground-truth outputs y_i. During training, we seek parameter values ϕ that minimize the loss and hence map the training inputs to the outputs as closely as possible.

2.4.1 Cross-Entropy Loss

The cross-entropy loss is based on the idea of finding parameters $\theta = (\mu, \sigma^2)$ that minimize the distance between the empirical distribution $q(y)$ of the observed data y and a model distribution $P_r(y|\theta)$ (figure 2.5). The distance between two probability distributions $q(z)$ and $p(z)$ can be evaluated using the Kullback-Leibler divergence (Section 1.1.1):

$$D_{KL}[q\|p] = \int_{-\infty}^{+\infty} q(z)\log[q(z)]dz - \int_{-\infty}^{+\infty} q(z)\log[p(z)]dz \tag{2.7}$$

Now consider that we observe an empirical data distribution at points $\{y_i\}_{i=1}^{I}$. We can describe this as a weighted sum of point masses:

$$q(y) = \frac{1}{I}\sum_{i=1}^{I}\delta\left[y - y_i\right] \tag{2.8}$$

where $\delta[\bullet]$ is the Dirac delta function (Section 1.1.2). We want to minimize the KL divergence between the model distribution $Pr(y|\theta)$ and this empirical distribution:

$$\begin{aligned}\hat{\theta} &= \underset{\theta}{\text{argmin}}\left[\int_{-\infty}^{+\infty} q(y)\log[q(y)]dy - \int_{-\infty}^{+\infty} q(y)\log[Pr(y|\theta)]dy\right] \\ &= \underset{\theta}{\text{argmin}}\left[-\int_{-\infty}^{+\infty} q(y)\log[Pr(y|\theta)]dy\right]\end{aligned} \tag{2.9}$$

where the first term disappears, as it has no dependence on θ. The remaining second term is known as the cross-entropy, which can be interpreted as the amount of uncertainty that remains in one distribution after taking into account what we already know from the other. Now, we substitute in the definition of $q(y)$ from equation 2.8:

$$\begin{aligned}\hat{\theta} &= \underset{\theta}{\text{argmin}}\left[-\int_{-\infty}^{+\infty}\left(\frac{1}{I}\sum_{i=1}^{I}\delta\left[y - y_i\right]\right)\log[Pr(y|\theta)]dy\right] \\ &= \underset{\theta}{\text{argmin}}\left[-\frac{1}{I}\sum_{i=1}^{I} log[Pr(y_i|\theta)]\right] \\ &= \underset{\theta}{\text{argmin}}\left[-\sum_{i=1}^{I} log[Pr(y_i|\theta)]\right]\end{aligned} \tag{2.10}$$

In machine learning, the distribution parameters θ are computed by the model $f[x_i, \phi]$, so we have:

$$\hat{\phi} = \underset{\phi}{\text{argmin}}\left[-\sum_{i=1}^{I} log\Big[Pr(y_i|f[x_i,\phi])\Big]\right] \tag{2.11}$$

with the softmax function (Section 1.1.3) as the probability:

$$Pr(y_i|f[x_i,\phi]) = \text{softmax}_{y_i}\Big[f[x_i,\phi]\Big] = \frac{\exp\Big[f_{y_i}[x_i,\phi]\Big]}{\sum_{k=1}^{I}\exp\Big[f_k[x_i,\phi]\Big]} \tag{2.12}$$

where $f_k[x,\phi]$ denotes the k^{th} output of the neural network.

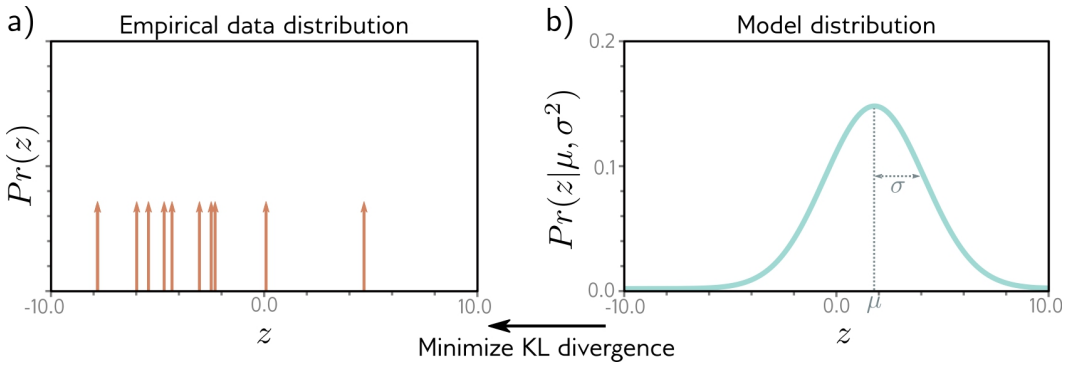


Figure 2.5: Cross-entropy method. a) Empirical distribution of training samples (arrows denote Dirac delta functions). b) Model distribution (a normal distribution with parameters $\theta = \mu, \sigma^2$).

2.4.2 Contrastive Learning

The idea behind contrastive learning is to learn a function that encodes the input into an embedding vector such that samples from the same class have similar embeddings while samples from different classes have very different ones. The most common usage of this idea is self-supervised contrastive learning, where an anchor gets compared to a single positive, which usually is an augmented version of the anchor (*i.e.*, a modified or transformed variant of the original sample, created through techniques like rotation, flipping, or scaling), and a set of negatives, consisting of the remainder of the batch.
A novel interpretation of this idea has been proposed by Khosla *et al.* called "Supervised Contrastive Learning" [KTW+20]. This approach contrasts the set of all samples from the same class as positives against the negatives from the remainder of the batch, as shown in Figure 2.6.

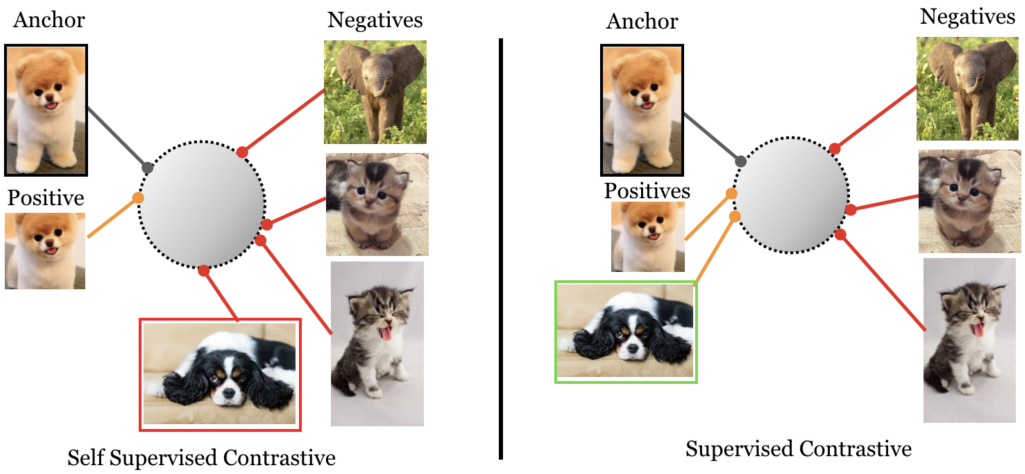


Figure 2.6: Supervised vs. self-supervise contrastive loss.

Self-supervised Contrastive Loss

Within a batch, let $i \in I \equiv 1, ..., N$ be the index of an arbitrary sample, and let $j(i)$ be the index of the augmented sample. In self-supervised contrastive learning, the loss takes the following form:

$$\mathcal{L}^{\text{self}} = \sum_{i \in I} \mathcal{L}_i^{\text{self}} = -\sum_{i \in I} \log \frac{\exp\left(v_i \cdot v_{j(i)}/\tau\right)}{\sum_{a \in A(i)} \exp\left(v_i \cdot v_a/\tau\right)} \tag{2.13}$$

where v_i represents the embedding of the input x_i, $\tau \in \mathbb{R}^+$ is a scalar temperature parameter and $A(i) \equiv I \backslash \{i\}$.

Supervised Contrastive Loss

Within a batch, let $i \in I \equiv 1, ..., N$ be the index of an arbitrary sample. In supervised contrastive learning, the loss takes the following form:

$$\mathcal{L}^{\text{sup}} = \sum_{i \in I} \mathcal{L}_i^{\text{sup}} = \sum_{i \in I} \frac{-1}{|P(i)|} \sum_{p \in P(i)} \log \frac{\exp\left(v_i \cdot v_p/\tau\right)}{\sum_{a \in A(i)} \exp\left(v_i \cdot v_a/\tau\right)} \tag{2.14}$$

where v_i represents the embedding of the input x_i, $\tau \in \mathbb{R}^+$ is a scalar temperature parameter, $A(i) \equiv I \backslash \{i\}$, $P(i) \equiv \{p \in A(i) : l_p = l_i\}$ is the set of indices of all positives in the batch apart from i (l_i is the label of x_i) and $|P(i)|$ is its cardinality. It uses the positive normalization factor (*i.e.*, $\frac{1}{|P(i)|}$) to remove bias present in multiple positives samples and preserve the summation over negatives in the denominator to increase performance.

2.5 Fitting models

Finding the parameters that minimize this loss as learning the network's parameters or simply as training or fitting the model. The process is to choose initial parameter values and then iterate the following two steps: (i) compute the derivatives (gradients) of the loss with respect to the parameters, and (ii) adjust the parameters based on the gradients to decrease the loss. After many iterations, we hope to reach the overall minimum of the loss function.

2.5.1 Gradient Descent

The goal of an optimization algorithm is to find parameters $\hat{\phi}$ that minimize the loss:

$$\hat{\phi} = \underset{\phi}{\text{argmin}}\Big[L[\phi]\Big] \tag{2.15}$$

The simplest method in this class is gradient descent. This starts with initial parameters $\phi = [\phi_0, \phi_1, ..., \phi_N]^T$ and iterates two steps:

- **Step 1.** Compute the derivatives of the loss with respect to the parameters:

$$\frac{\partial L}{\partial \phi} = \begin{bmatrix} \frac{\partial L}{\partial \phi_0} \\ \frac{\partial L}{\partial \phi_1} \\ \vdots \\ \frac{\partial L}{\partial \phi_N} \end{bmatrix} \tag{2.16}$$

- **Step 2.** Update the parameters according to the rule:

$$\phi \longleftarrow \phi - \alpha \cdot \frac{\partial L}{\partial \phi} \tag{2.17}$$

where the positive scalar α determines the magnitude of the change. The parameter α may be fixed (in which we call it a learning rate), or we may perform a line search where we try several values of α to find the one that most decreases the loss. At the minimum of the loss function the gradient will be zero, and the parameters will stop changing. In practice, we monitor the gradient magnitude and terminate the algorithm when it becomes too small.

2.5.2 Stochastic Gradient Descent

Using gradient descent to find the global optimum of a high-dimensional loss function is challenging. We can find a minimum, but there is no way to tell whether this is the global minimum or even a good one. One of the main problems is that the final destination of a gradient descent algorithm is entirely determined by the starting point. Stochastic gradient descent (SGD) attempts to remedy this problem by adding some noise to the gradient at each step.
The mechanism for introducing randomness is simple. At each iteration, the algorithm chooses a random subset of the training data and computes the gradient from these examples alone. The update rule for the model parameters ϕ_t at iteration t is hence:

$$\phi_{t+1} \longleftarrow \phi_t - \alpha \cdot \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i \left[\phi_t\right]}{\partial \phi} \tag{2.18}$$

where $\mathcal{B}_t$ is a set containing the indices of the input/output pairs in the current batch and, as before, l_i is the loss due to the i^{th} pair. The algorithm works through the training examples until it has used all the data, at which point it starts sampling from the full training dataset again. A single pass through the entire training dataset is referred to as an epoch.
SGD has several attractive features:

1. although it adds noise to the trajectory, it still improves the fit to a subset of the data at each iteration. Hence, the updates tend to be sensible even if they are not optimal.

2. because it draws training examples without replacement and iterates through the dataset, the training examples all still contribute equally.

3. it is less computationally expensive to compute the gradient from just a subset of the training data.

4. it can (in principle) escape local minima.

5. it reduces the chances of getting stuck near saddle points

There is also some evidence that SGD finds parameters for neural networks that cause them to generalize well to new data in practice.

2.5.3 Momentum

A common modification to stochastic gradient descent is to add a momentum term. We update the parameters with a weighted combination of the gradient computed from the current batch and the direction moved in the previous step:

$$\begin{aligned} m_{t+1} &\longleftarrow \beta \cdot m_t + (1-\beta) \sum_{i \in B_t} \frac{\partial l_i \left[\phi_t\right]}{\partial \phi} \\ \phi_{t+1} &\longleftarrow \phi_t - \alpha \cdot m_{t+1} \end{aligned} \tag{2.19}$$

where m_t is the momentum (which drives the update at iteration t), $\beta \in [0, 1)$ controls the degree to which the gradient is smoothed over time, and α is the learning rate.
The recursive formulation of the momentum calculation means that the gradient step is an infinite weighted sum of all the previous gradients, where the weights get smaller as we move back in time. The effective learning rate increases if all these gradients are aligned over multiple iterations but decreases if the gradient direction repeatedly changes as the terms in the sum cancel out. The overall effect is a smoother trajectory and reduced oscillatory behavior in valleys.

2.6 Residual Neural Networks

Residual neural networks (ResNets) are an innovative type of deep neural network architecture that introduces residual blocks, proposed by He *et al.* [HZRS16].
Every network that came before this processed the data sequentially, *i.e.* each layer receives the previous layer's output and passes the result to the next. In principle, we could add as many layers as we wanted to, however image classification performance decreased as further layers were added. To solve this, ResNets were presented. Here, each network layer computes an additive change to the current representation instead of transforming it directly. This allows deeper networks to be trained and improve performance across a variety of tasks.
In ResNets, each residual block contains a batch normalization operation, a ReLU activation function, and a convolutional layer.

2.6.1 Residual Connections and Residual Blocks

Let's focus on a local part of a neural network, as depicted in Figure 2.7. Denote the input by x. We assume that $f(x)$, the desired underlying mapping we want to obtain by learning, is to be used as input to the activation function on the top. On the left, the portion within the dotted-line box must directly learn $f(x)$. On the right, the portion within the dotted-line box needs to learn the residual mapping $g(x) = f(x) - x$, which is how the residual block derives its name. If the identity mapping $f(x) = x$ is the desired underlying mapping, the residual mapping amounts to $g(x) = 0$ and it is thus easier to learn: we only need to push the weights and biases of the upper weight layer (e.g., fully connected layer and convolutional layer) within the dotted-line box to zero. The right figure illustrates the residual block of ResNet, where the solid line carrying the layer input x to the addition operator is called a residual connection (or shortcut connection). With residual blocks, inputs can forward propagate faster through the residual connections across layers. In fact, the residual block can be thought of as a special case of the multi-branch Inception block: it has two branches one of which is the identity mapping.

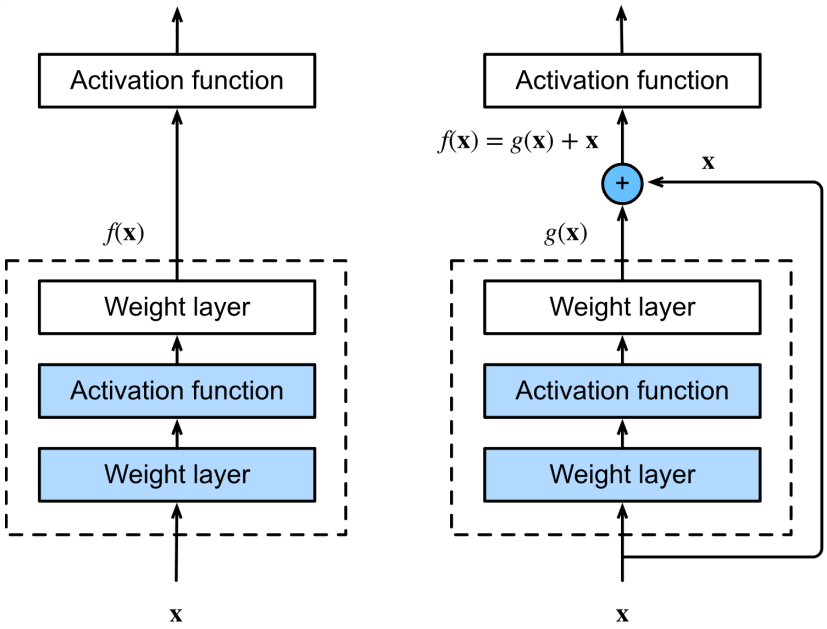


Figure 2.7: In a regular block (left), the portion within the dotted-line box must directly learn the mapping $f(x)$. In a residual block (right), the portion within the dotted-line box needs to learn the residual mapping $g(x) = f(x) - x$, making the identity mapping $f(x) = x$ easier to learn.

2.6.2 Batch Normalization

Batch normalization or BatchNorm is a technique, introduced by Sergey Ioffe and Christian Szegedy [IS15], to mitigate the internal covariate shift problem, *i.e.* the change in the distribution of network activations due to the change in network parameters during training.
Standardizing our data before entering training is essential for any model we're building and BatchNorm makes sure data continues to stay scaled while it's training.
Since the full whitening of each layer's inputs is costly and not everywhere differentiable, they make two necessary simplifications:

1. they normalize each scalar feature independently, by making it have the mean of 0 and the variance of 1. Note that simply normalizing each input of a layer may change what the layer can represent, so, to address this, they make sure that the transformation inserted in the network can represent the identity transform by introducing a pair of learnable parameters λ and β which scale and shift the normalized value.

2. in the batch setting where each training step is based on the entire training set, we would use the whole set to normalize activations, which is impractical when using stochastic optimization. To simplify this, since mini-batches are used in stochastic gradient training, each mini-batch produces estimates of the mean and variance of each activation. This way, the statistics used for normalization can fully participate in the gradient backpropagation.

The full algorithm for Batch normalization is presented below.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

2.6.3 Convolutional Layer

A convolutional layer comprises a set of filters (or kernels), smaller than the input image, with trainable parameters. During training, each filter slides across the height and width of the image, computing the dot product with the input at every spatial position. This process generates an activation map for each filter. In the convolution process (shown in Figure 2.8), the first entry of the activation map corresponds to the convolution of the filter with a specific region in the input image. This pattern repeats for every element in the input, producing the complete activation map. The output volume of the convolutional layer is formed by stacking the activation maps of all filters along the depth dimension. Each component of an activation map can be seen as the output of a neuron. Neurons are locally connected to small regions in the input image, with the region's size matching the filter size. Moreover, all neurons within an activation map share parameters. This local connectivity forces the network to learn filters that respond maximally to specific local regions in the input. The convolutional layers' design dictates that initial layers capture low-level features like lines, while subsequent layers extract higher-level features such as shapes and specific objects. This hierarchical approach enables the network to progressively learn and represent complex features in images. A padding may be applied to the input image to prevent it from reducing its dimensions after applying the filter.

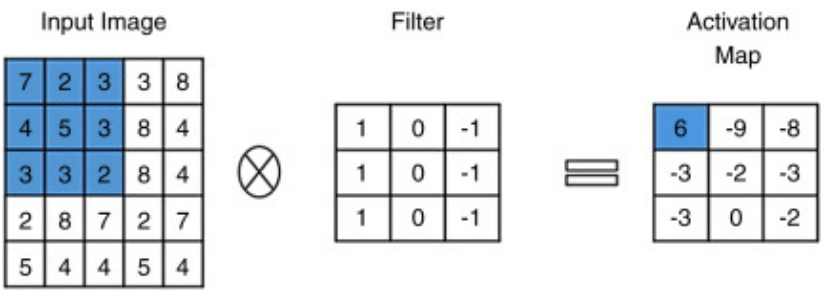


Figure 2.8: A graphical example of the convolution process.

2.6.4 ResNet18

ResNet18 is the 18 layer version of the Residual Neural Network proposed by He *et al.* [HZRS16]. The base version takes as input a 224*224*3 image, *i.e.* a 224*224 RGB image. The first layer is a Convolutional Layer with 64 kernels 7x7 with a stride of 2 and a padding of 3. This changes the dimension of the input to a 112*112*64. Then, a maxpool with kernel 3x3, a stride of 2 and padding of 1 is applied, which halves the dimension of the input again, making it a 56*56*64. The Network is divided in 4 main groups (Layer $1 \sim 4$ in Figure 2.9).

- **Layer 1.** Four Residual Blocks are applied to the network, with two skip connections after the second and the fourth. Each block has a convolutional layer with 64 3x3 kernel, a stride of 1, and a padding of 1. After applying all the Blocks, our input will maintain its dimension of 56*56*64.
- **Layer 2/3/4.** Four Residual Blocks are applied to the network, with two skip connections after the second and the fourth. Each block has a convolutional layer with 128/256/512 3x3 kernels, the first one with a stride of 2 and a padding of 1, while the other three with a stride of 1 and a padding of 1. The first skip connection will also feature an identity downsample (*i.e.*, a convolutional layer of 128/256/512 1x1 kernels, with stride of 2 and no padding, followed by a BatchNorm, that reduces the dimension of the input to be able to apply the sum after the skip connection). After applying all the Blocks, our input will have its dimension reduced to 28*28*128/14*14*256/7*7*512.

Finally, an avgpool is applied to the output of Layer 4, to obtain as output a vector of length 512. The last linear layer is optional, depending if want we want our output to be an embedding (no linear layer) or a classification.

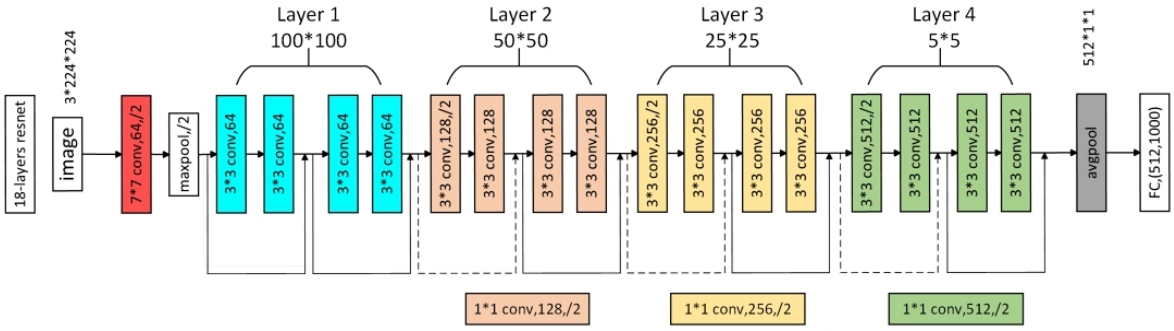


Figure 2.9: A visual explanation of the ResNet18 architecture.

Chapter 3

Linux Malware Analysis

Malware analysis is the art of dissecting malware to understand how it works, how to identify it, and how to defeat or eliminate it. Malicious software, or malware, plays a part in most computer intrusion and security incidents. Any software that does something that causes harm to a user, computer, or network can be considered malware, including viruses, trojan horses and worms. While the various malware incarnations do all sorts of different things, there exist a core set of tools and techniques at our disposal for analyzing malware.
To be able to properly explain how Malware Analysis works, we will use the books "Practical Malware Analysis" by Michael Sikorski and Andrew Honig [SH12] and "Practical Binary Analysis" by Dennis Andriesse [And18] [Pri24] as reference.

3.1 Type of Malware

There are various categories of malware, each with its own characteristics and methods of attack. Here are some common malware categories:

- **Backdoor** - Malicious code that installs itself onto a computer to allow the attacker access. Backdoors usually let the attacker connect to the computer with little or no authentication and execute commands on the local system.
- **Botnet** - Similar to a backdoor, in that it allows the attacker access to the system, but all computers infected with the same botnet receive the same instructions from a single command-and-control server.
- **Downloader** - Malicious code that exists only to download other malicious code. Downloaders are commonly installed by attackers when they first gain access to a system. The downloader program will download and install additional malicious code.
- **Information-stealing malware** - Malware that collects information from a victim's computer and usually sends it to the attacker. Examples include sniffers, password hash grabbers, and keyloggers. This malware is typically used to gain access to online accounts such as email or online banking.
- **Launcher** - Malicious program used to launch other malicious programs. Usually, launchers use nontraditional techniques to launch other malicious programs in order to ensure stealth or greater access to a system.

- **Rootkit** - Malicious code designed to conceal the existence of other code. Rootkits are usually paired with other malware, such as a backdoor, to allow remote access to the attacker and make the code difficult for the victim to detect.

- **Scareware** - Malware designed to frighten an infected user into buying something. It usually has a user interface that makes it look like an antivirus or other security program. It informs users that there is malicious code on their system and that the only way to get rid of it is to buy their "software," when in reality, the software it's selling does nothing more than remove the scareware.

- **Spam-sending malware** - Malware that infects a user's machine and then uses that machine to send spam. This malware generates income for attackers by allowing them to sell spam-sending services.

- **Worm or virus** - Malicious code that can copy itself and infect additional computers.

Malware often spans multiple categories. For example, a program might have a keylogger that collects passwords and a worm component that sends spam.
Malware can also be classified based on whether the attacker's objective is mass or targeted. Mass malware, such as scareware, takes the shotgun approach and is designed to affect as many machines as possible. Of the two objectives, it's the most common, and is usually the less sophisticated and easier to detect and defend against because security software targets it. Targeted malware, like a one-of-a-kind backdoor, is tailored to a specific organization. Targeted malware is a bigger threat to networks than mass malware, because it is not widespread and your security products probably won't protect you from it. Without a detailed analysis of targeted malware, it is nearly impossible to protect your network against that malware and to remove infections. Targeted malware is usually very sophisticated, and your analysis will often require the advanced analysis skills covered in this book.

3.2 Binary Analysis

Binary analysis is the science of analyzing the properties of binary computer programs, called binaries, and the machine code and data they contain. The goal of binary analysis is to figure out the true properties and behaviours of binary programs. Broadly speaking, you can divide binary analysis techniques into two classes:

- **Static analysis** - Static analysis techniques reason about a binary without running it. This approach has several advantages: you can potentially analyze the whole binary in one go, and you don't need a CPU that can run the binary. The downside is that static analysis has no knowledge of the binary's runtime state, which can make the analysis very challenging.

- **Dynamic analysis** - In contrast, dynamic analysis runs the binary and analyzes it as it executes. This approach is often simpler than static analysis because you have full knowledge of the entire runtime state, including the values of variables and the outcomes of conditional branches. However, you see only the executed code, so the analysis may miss interesting parts of the program.

3.2.1 Challenges

Binary analysis much more difficult than equivalent analysis at the source code level. Some of the challenges that characterize it are:

- **No symbolic information** - When we write source code in a high-level language like C or C++, we give meaningful names to constructs such as variables, functions, and classes. We call these names symbolic information, or symbols for short. Good naming conventions make the source code much easier to understand, but they have no real relevance at the binary level. As a result, binaries are often stripped of symbols, making it much harder to understand the code.
- **No type information** - Another feature of high-level programs is that they revolve around variables with well-defined types, such as int, float, or string, as well as more complex data structures like struct types. In contrast, at the binary level, types are never explicitly stated, making the purpose and structure of data hard to infer.
- **No high-level abstractions** - Modern programs are compartmentalized into classes and functions, but compilers throw away these high-level constructs. That means binaries appear as huge blobs of code and data, rather than well-structured programs, and restoring the high-level structure is complex and error-prone.
- **Mixed code and data** - Binaries can (and do) contain data fragments mixed in with the executable code. This makes it easy to accidentally interpret data as code, or vice versa, leading to incorrect results.
- **Location-dependent code and data** - Because binaries are not designed to be modified, even adding a single machine instruction can cause problems as it shifts other code around, invalidating memory addresses and references from elsewhere in the code. As a result, any kind of code or data modification is extremely challenging and prone to breaking the binary.

3.2.2 Anatomy of a Binary

Binaries are produced through compilation, which is the process of translating human-readable source code, such as C or C++, into machine code that your processor can execute. Figure 3.1 shows the steps involved in a typical compilation process for C code. Compiling C code involves four phases: preprocessing, compilation, assembly, and linking.

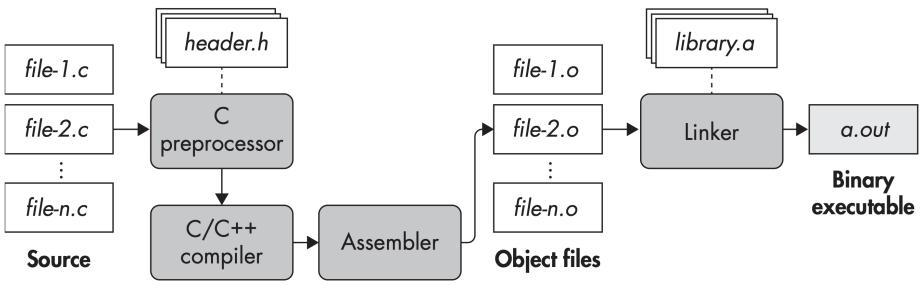


Figure 3.1: The C compilation process.

The Preprocessing Phase

The compilation process starts with a number of source files that you want to compile (shown as file-1.c through file-n.c in Figure 3.1). It's possible to have just one source file, but large programs are typically composed of many files. Not only does this make the project easier to manage, but it speeds up compilation because if one file changes, you only have to recompile that file rather than all of the code.
C source files contain macros (denoted by #define) and #include directives. You use the #include directives to include header files (with the extension .h) on which the source file depends. The preprocessing phase expands any #define and #include directives in the source file so all that's left is pure C code ready to be compiled.

The Compilation Phase

After the preprocessing phase is complete, the source is ready to be compiled. The compilation phase takes the preprocessed code and translates it into assembly language.
The reason why the compilation phase produces assembly language and not machine code is because, due to the high number of popular compiled languages (like C, C++, Objective-C, Common Lisp, Delphi, Go, and Haskell), writing a compiler that directly emits machine code for each of these languages would be an extremely demanding and time-consuming task. It's better to instead emit assembly code and have a single dedicated assembler that can handle the final translation of assembly to machine code for every language.
So, the output of the compilation phase is assembly, in reasonably human-readable form, with symbolic information intact.

The Assembly Phase

The input of the assembly phase is the set of assembly language files generated in the compilation phase, and the output is a set of object files, sometimes also referred to as modules. Object files contain machine instructions that are in principle executable by the processor.

The Linking Phase

The linking phase is the final phase of the compilation process. As the name implies, this phase links together all the object files into a single binary executable.
The program that performs the linking phase is called a linker, or link editor and it's typically separate from the compiler, which usually implements all the preceding phases.
Object files are compiled independently from each other, preventing the compiler from assuming that an object will end up at any particular base address. Moreover, object files may reference functions or variables in other object files or in libraries that are external to the program. Before the linking phase, the addresses at which the referenced code and data will be placed are not yet known, so the object files only contain relocation symbols that specify how function and variable references should eventually be resolved. In the context of linking, references that rely on a relocation symbol are called symbolic references. When an object file references one of its own functions or variables by absolute address, the reference will also be symbolic.
The linker's job is to take all the object files belonging to a program and merge them into a single coherent executable, typically intended to be loaded at a particular memory address.

3.2.3 Symbols and Stripped Binaries

High-level source code, such as C code, centers around functions and variables with meaningful, human-readable names. When compiling a program, compilers emit symbols, which keep track of such symbolic names and record which binary code and data correspond to each symbol. For instance, function symbols provide a mapping from symbolic, high-level function names to the first address and the size of each function. This information is normally used by the linker when combining object files and also aids debugging.
Unfortunately, extensive debugging information typically isn't included in production-ready binaries, and even basic symbolic information is often stripped to reduce file sizes and prevent reverse engineering, especially in the case of malware or proprietary software.

3.2.4 The ELF Format

Executable and Linkable Format (ELF) is the default binary format on Linux-based systems and is used for executable files, object files, shared libraries, and core dumps.
Every ELF file starts with an executable header, which is just a structured series of bytes telling you that it's an ELF file, what kind of ELF file it is, and where in the file to find all the other contents.
The code and data in an ELF binary are logically divided into contiguous non-overlapping chunks called sections. Sections don't have any predetermined structure and are described by a section header, which denotes the properties of the section and allows you to locate the bytes belonging to the section. The section headers for all sections in the binary are contained in the section header table.
The program header table provides a segment view of the binary, as opposed to the section view provided by the section header table. The section view of an ELF binary is meant for static linking purposes only. In contrast, the segment view is used by the operating system and dynamic linker when loading an ELF into a process for execution to locate the relevant code and data and decide what to load into virtual memory.

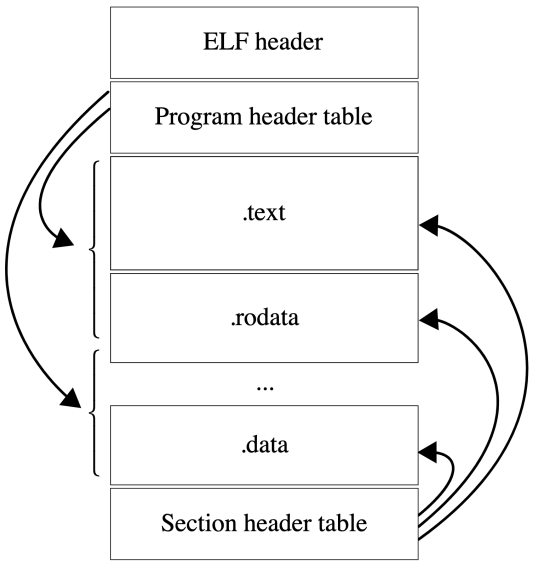


Figure 3.2: Structure of an ELF file.

3.3 Static Analysis

Static analysis describes the process of analyzing the code or structure of a program to determine its function. The program itself is not run at this time in contrast of when performing dynamic analysis.

3.3.1 Antivirus Scanning

When first analyzing prospective malware, a good first step is to run it through multiple antivirus programs, which may already have identified it. But antivirus tools are certainly not perfect. They rely mainly on a database of identifiable pieces of known suspicious code (file signatures), as well as behavioral and pattern-matching analysis (heuristics) to identify suspect files. One problem is that malware writers can easily modify their code, thereby changing their program's signature and evading virus scanners. Also, rare malware often goes undetected by antivirus software because it's simply not in the database. Finally, heuristics, while often successful in identifying unknown malicious code, can be bypassed by new and unique malware.

3.3.2 Hashing

Hashing is a common method used to uniquely identify malware. The malicious software is run through a hashing program that produces a unique hash that identifies that malware. The most commonly used hash functions are Message-Digest Algorithm 5 (MD5), Secure Hash Algorithm 1 (SHA-1) and Secure Hash Algorithm 256 (SHA-256) (Section 1.4). Once you have a unique hash for a piece of malware, you can use it as a label and to see if the file has already been identified online.

3.3.3 Linked Libraries and functions

One of the most useful pieces of information that we can gather about an executable is the list of functions that it imports. Imports are functions used by one program that are actually stored in a different program, such as code libraries that contain functionality common to many programs. Code libraries can be connected to the main executable by linking.
Programmers link imports to their programs so that they don't need to re-implement certain functionality in multiple programs. Code libraries can be linked statically, at runtime, or dynamically.

- **Static Linking** - When a library is statically linked to an executable, all code from that library is copied into the executable, which makes the executable grow in size. When analyzing code, it's difficult to differentiate between statically linked code and the executable's own code.
- **Runtime Linking** - Executables that use runtime linking connect to libraries only when that function is needed, not at program start, as with dynamically linked programs. It is commonly used in malware, especially when it's packed or obfuscated.
- **Dynamic Linking** - When libraries are dynamically linked, the host OS searches for the necessary libraries when the program is loaded. When the program calls the linked library function, that function executes within the library.

3.4 Disassembly

When people say "disassembly" they usually mean static disassembly, which involves extracting the instructions from a binary without executing it. In contrast, dynamic disassembly, more commonly known as execution tracing, logs each executed instruction as the binary runs. The goal of every static disassembler is to translate all code in a binary into a form that a human can read or a machine can process (for further analysis).
There are two major approaches to static disassembly, each of which tries to avoid disassembly errors in its own way: linear disassembly and recursive disassembly.

3.4.1 Linear Disassembly

Linear disassembly is conceptually the simplest approach. It iterates through all code segments in a binary, decoding all bytes consecutively and parsing them into a list of instructions. The risk of using linear disassembly is that not all bytes may be instructions. For example, some compilers, such as Visual Studio, intersperse data such as jump tables with the code, without leaving any clues as to where exactly that data is. If disassemblers accidentally parse this inline data as code, they may encounter invalid opcodes, *i.e.* the portion of a machine language instruction that specifies the operation to be performed.

3.4.2 Recursive Disassembly

Unlike linear disassembly, recursive disassembly is sensitive to control flow. It starts from known entry points into the binary and from there recursively follows control flow (such as jumps and calls) to discover code. This allows recursive disassembly to work around data bytes in all but a handful of corner cases. The downside of this approach is that not all control flow is so easy to follow. For instance, it's often difficult, if not impossible, to statically figure out the possible targets of indirect jumps or calls.
Recursive disassembly is the de facto standard in many reverse-engineering applications, such as malware analysis.

3.4.3 Structuring Code

There are various ways of structuring disassembled code that make the code easier to analyze in two ways:

- **Compartmentalizing**: By breaking the code into logically connected chunks, it becomes easier to analyze what each chunk does and how chunks of code relate to each other.
- **Revealing control flow**: Some of the code structures I'll discuss next explicitly represent not only the code itself but also the control transfers between blocks of code. These structures can be represented visually, making it much easier to quickly see how control flows through the code and to get a quick idea of what the code does.

Functions

In most high-level programming languages, functions are the fundamental building blocks used to group logically connected pieces of code. Programs that are well structured and

properly divided into functions are much easier to understand than poorly structured programs. For this reason, most disassemblers make some effort to recover the original program's function structure and use it to group disassembled instructions by function. This is known as function detection.
For binaries with symbolic information, function detection is trivial: the symbol table specifies the set of functions, along with their names, start addresses, and sizes. Unfortunately, many binaries are stripped of this information, which makes function detection far more challenging.
The predominant strategy that disassemblers use for function detection is based on function signatures, which are patterns of instructions often used at the start or end of a function. This strategy is used in all well-known recursive disassemblers.

Control-Flow Graphs

Breaking the disassembled code into functions is one thing, but some functions are quite large, which means analyzing even one function can be a complex task. To organize the internals of each function, disassemblers and binary analysis frameworks use another code structure, called a control-flow graph (CFG).

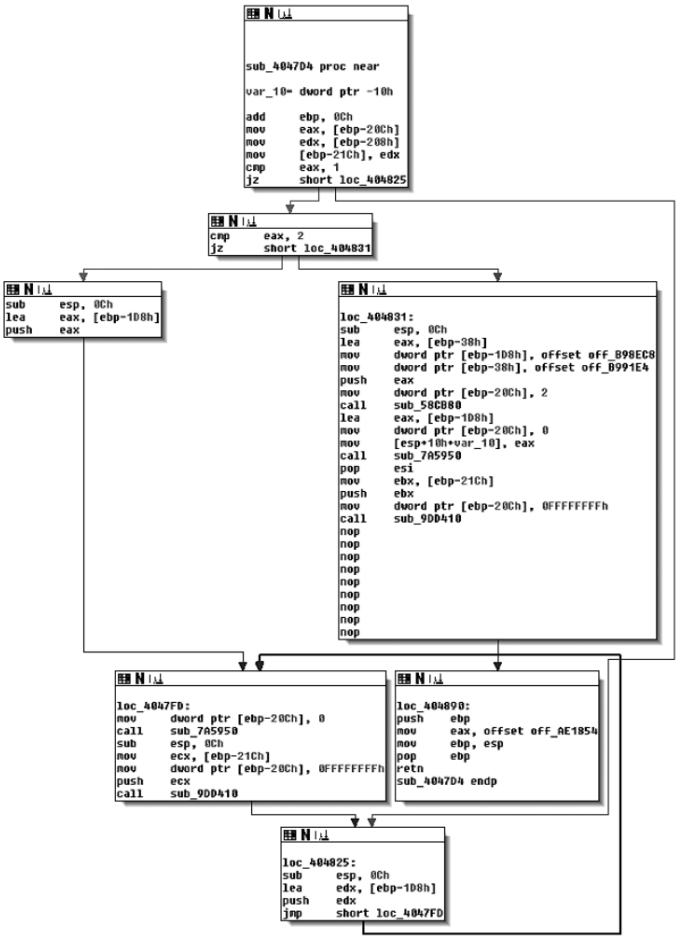


Figure 3.3: A CFG as seen in IDA Pro.

As you can see in Figure 3.3, CFGs represent the code inside a function as a set of code blocks, called basic blocks, connected by branch edges, shown here as arrows. A basic block is a sequence of instructions, where the first instruction is the only entry point (the only instruction targeted by any jump in the binary), and the last instruction is the only exit point (the only instruction in the sequence that may jump to another basic block).
Call edges are not part of a CFG because they target code outside of the function. There is another code structure, called a call graph, that is designed to represent the edges between call instructions and functions.

Call Graphs

Call graphs are similar to CFGs, except they show the relationship between call sites and functions rather than basic blocks. In other words, CFGs show you how control may flow within a function, while call graphs show you which functions may call each other.

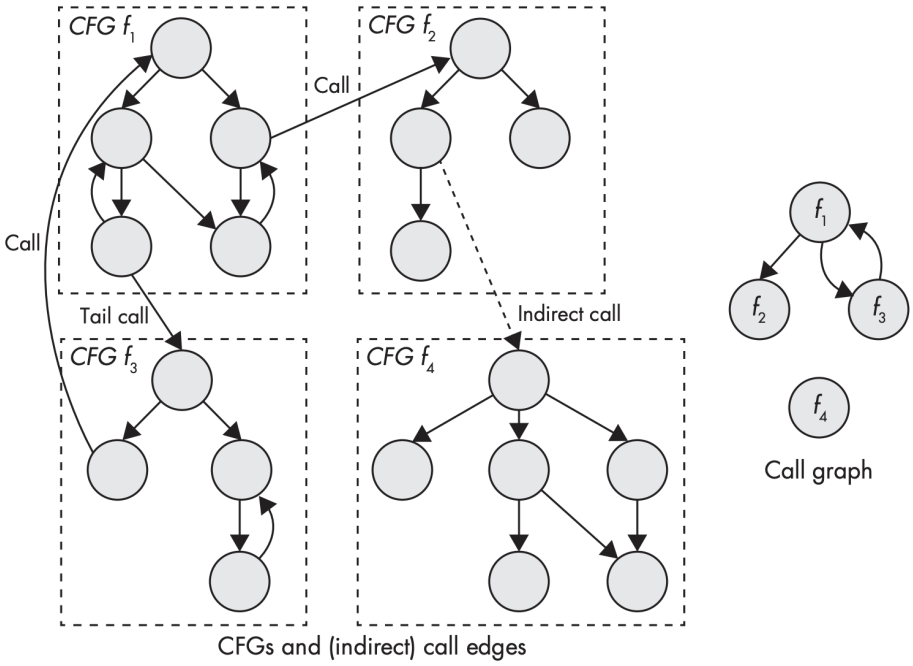


Figure 3.4: CFGs and connections between functions (left) and the corresponding call graph (right).

3.5 Packed and Obfuscated Malware

Malware writers often use packing or obfuscation to make their files more difficult to detect or analyze. Obfuscated programs are ones whose execution the malware author has attempted to hide. Packed programs are a subset of obfuscated programs in which the malicious program is compressed and cannot be analyzed. Both techniques will severely limit your attempts to statically analyze the malware.

3.5.1 Obfuscation Techniques

To explain the different existing obfuscation techniques we will use the article by Rabia Tahir "A Study on Malware and Malware Detection Techniques" [Tah18] and the paper by You and Yim "Malware Obfuscation Techniques: A Brief Survey" [YY10].

Dead-Code Insertion

This is easiest way to change code without affecting its meaning. In this technique, garbage code or statements are inserted in the code by using NOP statements and push followed by pop. These statements are used in such a sequence that it does not affect the semantics of code, while making the detection harder.

Instruction Replacement

In this technique, instructions are replaced with other instructions that generate the same meaning just like synonyms in natural languages. For example all following instructions have the same effect on register eax: set the register value to zero.

```
move eax,0
xor eax,eax
and eax,0
sub eax,eax
```

Since instruction are substituted with equivalent ones, this technique makes the detection of a malware very hard.

Register Reassignment

This technique reassign register in every copy without changing the semantics of the virus. It is a simple technique but when combine with other technique can make detection very difficult.

Subroutine Reordering

A set of instructions in a piece of code is permuted so that the code changes in its appearance while keeping the same behavior. An example of a subroutine followed by its reordering is shown below:

```
//Subroutine
mov eax,0A
push ecx
add esi,ebx
//Reordering
add esi,ebx
mov eax,0A
push ecx
```

Code Transposition

Code transposition reorders the sequence of the instructions of an original code without having any impact on its behavior. There are two methods to achieve this technique.
The first method randomly shuffles the instructions, and then recovers the original execution order by inserting the unconditional branches or jumps. It is not difficult to defeat this method because the original program can be easily restored by removing the unconditional branches or jumps. On the other hand, the second method creates new generations by choosing and reordering the independent instructions that have no impact on one another. Because it is a complex problem to find the independent instructions, this method is hard to implement, but can make the cost of detection high.

Code Integration

In code integration a malware knits itself to the code of its target program. In order to apply this technique, the target program is first decompiled into manageable objects, the malware is then seamlessly added between them, and then the integrated code is reassembled into a new generation. As one of the most sophisticated obfuscation techniques, code integration can make detection and recovery very difficult.

3.5.2 Packing Files

When the packed program is run, a small wrapper program also runs to decompress the packed file and then run the unpacked file, as shown in Figure 3.5. When a packed program is analyzed statically, only the small wrapper program can be dissected.

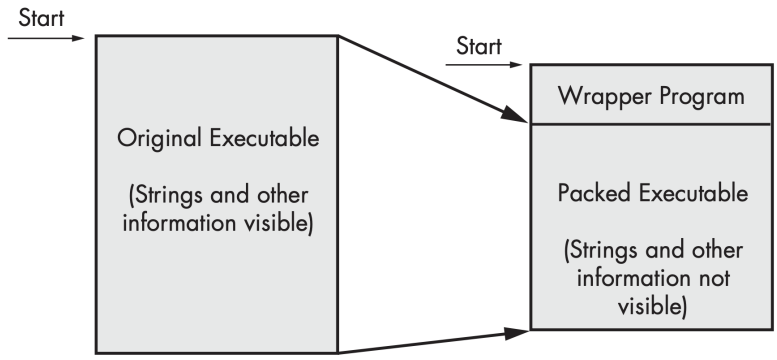


Figure 3.5: The file on the left is the original executable, with all strings, imports, and other information visible. On the right is a packed executable. All of the packed file's strings, imports, and other information are compressed and invisible to most static analysis tools.

Chapter 4

Robust Android Malware Familial Classification

Our first objective for this thesis is to test the results obtained by Wu *et al.* in their paper "Contrastive Learning for Robust Malware Familial Classification" [WDZ+22].
Once thoroughly explained how the system works, we'll show how we were able to replicate it by presenting the codes written and some intermediate results to better explain the process. Once this is done, in the next chapter the focus will shift on applying the ideas presented in the paper on a different environment: Linux IoT.

4.1 Motivation

Being the most widely used mobile operating system and due to its open-source nature and market openness, Android has been the main target of malware attacks for years [Kiv23].
To be able to hide their malicious tasks, different code obfuscation techniques have been applied by attackers [DLD+18]. After obfuscation, malware samples become more difficult to analyze and even samples from the same family may appear different from each other if different obfuscation techniques have been applied.
Most traditional Android malware analysis methods cannot resist code obfuscation. Android malware familial classifiers can be mainly divided in two categories: string-based and graph-based. The first methods usually focus on permission requested by apps and search for the presence of strings (*e.g.*, API calls) inside the disassembled code to build models to analyze Android malware, but can be usually easily evaded by obfuscation due to their lack of structural and contextual information of the program behaviour. On the other hand, graph-based methods distill the program semantics of apps into graph representations and apply graph matching to analyze the malware families. However, as shown by Dong *et al.* [DLD+18], more often than not malware creators perform complex obfuscations (*e.g.*, control-flow obfuscation) to hide their malicious objectives, hindering the results of the models by rendering useless many of the features extracted with these methods.
To address this issue, Wu *et al.* [WDZ+22] propose the use of contrastive learning, due to its powerful high level feature extraction and its many successful applications in other fields (*e.g.*, text representation learning and language understanding).

4.2 System Architecture

In this section we'll show how *IFDroid*, the contrastive learning-based robust and interpretable Android malware classification system proposed by Wu *et al.* [WDZ+22], works.

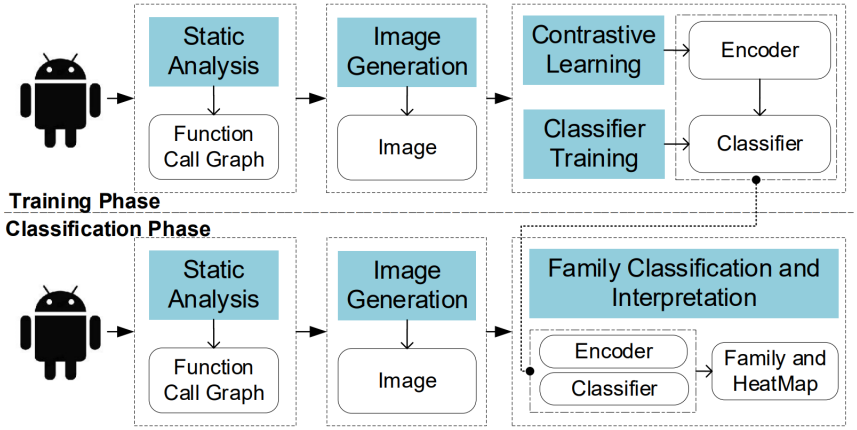


Figure 4.1: System Overview of *IFDroid*.

As shown in Figure 4.1, *IFDroid* consists of two main phases: the Training Phase and the Classification Phase.
During the Training Phase we aim to train a robust encoder (via Contrastive Learning) and a classifier. To do so we follow four steps:

1. **Static Analysis**: the Function Call Graph (FCG) is extracted from the malware using a software called Androguard [Des11] where every node is either an API call or a user-defined function (UDF).

2. **Image Generation**: the FCG is transformed into an image using centrality measures as pixels.

3. **Contrastive Learning**: an encoder based on a ResNet18 Neural Network with a SupConLoss is trained using the generated images as input.

4. **Classifier Training**: a classifier is trained using the encoded images as input.

During the Classification Phase we use the trained model to classify unlabeled malware into their corresponding families.

4.2.1 Static Analysis

It has been empirically demonstrated that graph representation is more robust than string-based features [FLL+18], therefore *IFDroid* performs low-cost program analysis (*e.g.*, context- and flow-insensitive analysis) to distill the program semantics of a malware sample into a function call graph. To do so, a widely used reverse engineering tool named Androguard [Des11] is used to perform the Static Analysis.

Figure 4.2: Androguard Logo.

4.2.2 Image Generation

Deep-learning based image classification has been object of research for several decades and can now process millions of images while maintaining high accuracy. Furthermore, its output can be visualized to give a better intuition to users. Because of these advantages, image-based methods have been widely employed in malware analysis. However, most of these approaches [NKJM11] use semantic-insensitive mapping algorithms (*e.g.*, reshaping the binary as a 0-1 Matrix), reducing the accuracy of the model.
To achieve efficient and semantic-sensitive malware analysis, Wu *et al.* propose the use of centrality analysis: a technique that checks the importance of a node in the graph by analyzing its centrality measures. For this model, four widely used centrality measures are selected:

1. **Degree Centrality** [F+02] : assigns an importance score solely based on the number of edges of a node. The value is then normalized by the maximum possible degree in a graph $N-1$.

$$C_d(i) = \frac{\deg(i)}{N-1} \tag{4.1}$$

2. **Katz Centrality** [Kat53] : assigns an importance score based on the sum of the neighbours scaled by a constant that decreases based the further the neighbour is. A^k is the adjacency matrix at distance k, while α is a constant between 0 and 1. The vector containing all the nodes' centralities is then normalized.

$$C_k(i) = \sum_{k=1}^{\infty} \sum_{j=1}^{+\infty} \alpha^k \left(A^k\right)_{ji} \tag{4.2}$$

3. **Closeness Centrality** [F+02] : assigns an importance score negatively correlated to the average of the shortest path length from the node to every other node in the graph normalized by the maximum possible path length in a graph $N-1$.

$$C_c(i) = \frac{N-1}{\sum_{i \neq j} d(i,j)} \tag{4.3}$$

4. **Harmonic Centrality** [ML00] : assigns an importance score based on the reciprocal of the harmonic mean of the distances from a node to all other nodes in the graph.

$$C_h(i) = \frac{\sum_{i \neq j} \frac{1}{d(i,j)}}{N-1} \tag{4.4}$$

To generate the image, for every node taken into consideration all four centrality measures are calculated and then written into a vector. That vector is then reshaped into a square grayscale image by padding the extra pixels with zeros (*i.e.*, black pixels).
The main objective now is how to select which nodes to take into consideration.

Sensitive APIs Selection

Android apps use API calls to access operating system functionality and system resources. On the other hand, malware samples tend to invoke sensitive APIs to perform malicious tasks. For example, *getDeviceID* can get your phone's IMEI and *getLine1Number* can obtain your phone number.
Gong *et al.* [GLQ+20] showed that, out of the >50k API calls provided by the current Android SDK (Software Developer Kit), there are 426 that have a high correlation to malicious behaviour. To find this subset, they used *Spearman's rank correlation coefficient* (SRC [Spe04]) to evaluate the statistical correlation between an API call and the malice of apps. Their API selection approach consists of four steps:

- **Step 1. Selecting APIs with the highest correlation with malware**: using SRC, it tracks the APIs with $|SRC| \geq 0.2$. Out of all of these, many of them are seldom invoked by the apps in our dataset (invoked by fewer than 0.1% apps). Using these infrequently invoked APIs as features may bring over-fitting problems to machine learning, and thus the API selector has to neglect these. After Step 1, we have a total of 260 APIs that satisfy the requirements.

- **Step 2. Selecting APIs that relate to restrictive permissions**: to protect the privacy/security of user information, an app needs to request permissions before obtaining certain information or fulfilling certain functions. Android permissions are classified into three protection levels: normal, signature, and dangerous. The APIs protected by dangerous-level permissions and those by signature level permissions are oftentimes relevant to sensitive user data (such as camera, SMS, and location data), which are thus crucial for malware detection. The API selector takes advantage of the Axplorer [BBD+16] and PScout [AZHL12] tools to select the APIs related to restrictive permissions. After Step 2, we have a total of 112 APIs that satisfy the requirements.

- **Step 3. Selecting APIs that perform sensitive operations**: the third strategy selects APIs that perform sensitive operations. there are five categories of sensitive operations commonly exploited for conducting attacks: (1) APIs that can lead to privilege escalation, (*e.g.*, shell command execution) APIs, (2) APIs for database operations and file read/write, which are commonly used in privacy leakage attacks, (3) APIs operating on key Android components, (*e.g.*, those for creating an Android window or overlay), which are used in attacks such as Activity hijacking, (4) cryptographic operation APIs, which are commonly used in ransomware attacks, and (5) APIs for dynamic code loading, which can load malicious payloads at runtime and perform attacks such as update attack. After Step 3, we have a total of 70 APIs that satisfy the requirements.

- **Step 4. Combining the above**: The last step is combining the above strategies, leading to a total of 426 key APIs (after removing duplicates).

4.2.3 Contrastive Learning

The goal of contrastive learning is to maximize the agreement between original data and its positive data while minimize the agreement between original data and its negative data by using a contrastive loss in the vector space.

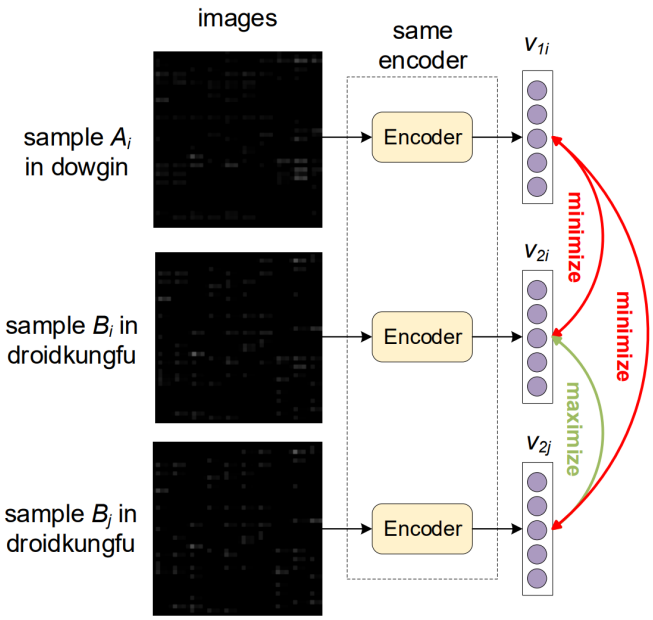


Figure 4.3: Supervised contrastive learning in *IFDroid*.

It was found by Khosla *et al.* [KTW+20] that the label information of samples can be used to improve the accuracy of contrastive learning, therefore supervised contrastive learning is selected to train *IFDroid*'s encoder.
The loss used to train the encoder (SupConLoss) calculates the contrastive loss in batches. Given an input batch of data, SupConLoss works as the following:

$$\mathcal{L}_{\text{out}}^{\text{sup}} = \sum_{i \in I} \frac{-1}{|P(i)|} \sum_{p \in P(i)} \log \frac{\exp\left(v_i \cdot v_p / \tau\right)}{\sum_{a \in A(i)} \exp\left(v_i \cdot v_a / \tau\right)} \tag{4.5}$$

where I is the set of indices of the inputs, v_i represents the embedding of the input x_i, $\tau \in \mathbb{R}^+$ is a scalar temperature parameter, $A(i) \equiv I \backslash \{i\}$, $P(i) \equiv \{p \in A(i) : l_p = l_i\}$ is the set of indices of all positives in the batch apart from i (l_i is the label of x_i) and $|P(i)|$ is its cardinality. It uses the positive normalization factor (*i.e.*, $\frac{1}{|P(i)|}$) to remove bias present in multiple positives samples and preserve the summation over negatives in the denominator to increase performance.

Parameters	Settings
loss function	SupConLoss in [KTW+20]
temperature	0.07
optimizer	SGD
momentum	0.9
weight decay	0.0001
learning rate	0.05
batch size	64
epoch	100

Table 4.1: Paramaters used in *IFDroid* contrastive learning

The authors chose ResNet18 [HZRS16] as the image encoder after experimenting with many widely-used neural networks, since it can achieve a balance between accuracy and efficiency. By applying centrality analysis on the 426 sensitive APIs discussed in the previous section, the input of *IFDroid* becomes a 42*42 pixel grayscale image (with the last 60 pixels black, since $42 * 42 - 426 * 4 = 60$). Since the inputs in most computer vision datasets are 224*224 pixels RGB images, to make ResNet18 suitable for the inputs generated by this model it needs to be modifies so that the dimension of the input is the same as the size of its image. Table 4.1 shows the details of parameters used in *IFDroid* contrastive learning. The loss function is the same as in [KTW+20], namely Supervised Contrastive Loss. The whole procedure is trained using Stochastic Gradient Descent (SGD) with 0.9 momentum and 0.0001 weight decay. The output in this step is a learned encoder that can convert an image into a vector whose dimension is 512.

4.2.4 Classifier Training

In this step, the trained encoder is used to obtain the inputs for the classifier, a one-layer fully connected neural network that uses Cross Entropy Loss to correctly sort the input malware sample into their label. The output of the classifier is a n-tuple where n is the number of different labels possible. The Cross Entropy Loss works as the following:

$$\mathcal{L}_{ce} = -\sum_{i \in I} \log \frac{\exp\left(x_{i,l_i}\right)}{\sum_{c \in C} \exp\left(x_{i,c}\right)} \tag{4.6}$$

where I is the set of indices of the inputs, C is the set of possible labels, $x_{i,c}$ is the value of the output for the label c and l_i is the correct label of the input x_i.

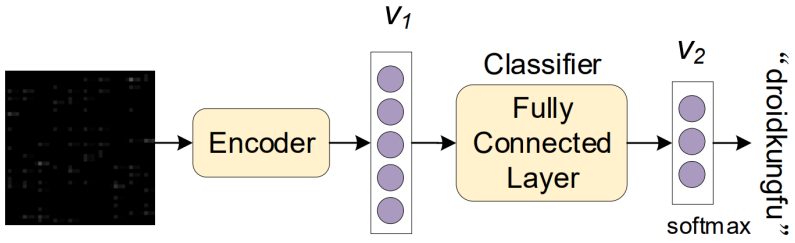


Figure 4.4: Familial classification of *IFDroid*.

4.3 Our Experiment

Before applying the ideas behind *IFDroid* to Linux IoT malware classification, we want to re-create the experiments performed by Wu *et al.* [WDZ+22], as to check the replicability of their results.

4.3.1 Obtaining the Malware Samples

To be able to train our neural network, we first need a Dataset to work on. Following the original paper, we land on AndroZoo [ABKLT16], a growing collection of Android Applications collected from several sources, including the official Google Play app market. It currently contains 24,264,358 different APKs[1], each of which has been (or will soon be) analysed by tens of different AntiVirus products to know which applications are detected as Malware [ABKLT24].

AndroZoo

This dataset presents a huge collection of APKs collected from different sources such as Google Play, Anzhi, AppChina and many others. It was created by using a dedicated web crawler using the scrapy framework, a fast high-level web crawling and scraping framework for Python. Every candidate app which was available for free run through a processing pipeline that:

1. Ensured that app had not already been downloaded;
2. Downloads the file;
3. Computes its SHA256 checksum;
4. Archives the file.

The obtained APKs are then classified using Euphony [HSTD+17], a tool to infer a single label of a malicious application from a list of VirusTotal reports.
After being granted access to the dataset, we obtained an API key. With this API key we were able to download any APK inside of the dataset simply providing its SHA256, which represents the hash value of a malware sample and is used to uniquely identify files, in the following way:

```
https://androzoo.uni.lu/api/download?apikey=${APIKEY}&sha256=${SHA256}
```

Using a simple Python script, iterating over the list of SHA256s provided by the AndroZoo team, we were able to download the needed samples.
Due to storage and time deficiency, we reduced our dataset to 5000 samples, equally divided between 5 malware families: Airpush, Droidkungfu, Feiwo, Utchi e Wooboo.
These malware families are all Adware, *i.e.* a type of malware that displays or downloads advertising material on the system, except for Droidkungfu, which is a Trojan malware that forwards confidential details to a remote server.

[1] as of 2024-02-07.

4.3.2 Extracting the Function Call Graph

After obtaining the samples, it's time to extract the Function Call Graph (FCG) from the APKs. To reach this objective, we'll use the same tool used for *IFDroid*, *i.e.* Androguard [Des11].

Androguard

Androguard is an open-source, Python-based security tool designed for analyzing and reverse engineering Android applications. The specific command we are going to use is "androguard cg", which creates a Call Graph from APK and exports it in .gml format, a simple text-based file format used to represent graph structures. The command works as the following:

```
Usage: androguard cg [OPTIONS] APK

  Create a call graph and export it into a graph format.

  The default is to create a file called callgraph.gml in
     the current directory!

  classnames are found in the type "Lfoo/bar/bla;".

  Example:

      $ androguard cg examples/tests/hello-world.apk

Options:
  -o, --output TEXT           Filename of the output file,
                              the extension is used to
                              decide which format to use
                              [default: callgraph.gml]
  -s, --show                  instead of saving the graph,
                              print it with mathplotlib
                              (you might not see anything!)
  -v, --verbose               Print more output
  --classname TEXT            Regex to filter by classname
                              [default: .*]
  --methodname TEXT           Regex to filter by methodname
                              [default: .*]
  --descriptor TEXT           Regex to filter by descriptor
                              [default: .*]
  --accessflag TEXT           Regex to filter by accessflags
                              [default: .*]
  --no-isolated / --isolated  Do not store methods which has
                              no xrefs
  --help                      Show this message and exit.
```

We use this tool on our 5000 samples to obtain their FCGs in .graphml format, which is more suitable than .gml for complex graphs like the ones we are creating. To better show the complexity of the graphs generated in this way, in Figure 4.5 is presented the FCG of the smallest Wooboo sample out of the 1000 we used, rendered using Gephi [BHJ09], a open-source software for the visualization and exploration of all kinds of graphs and networks, using the Fruchterman Reingold layout [FR91].

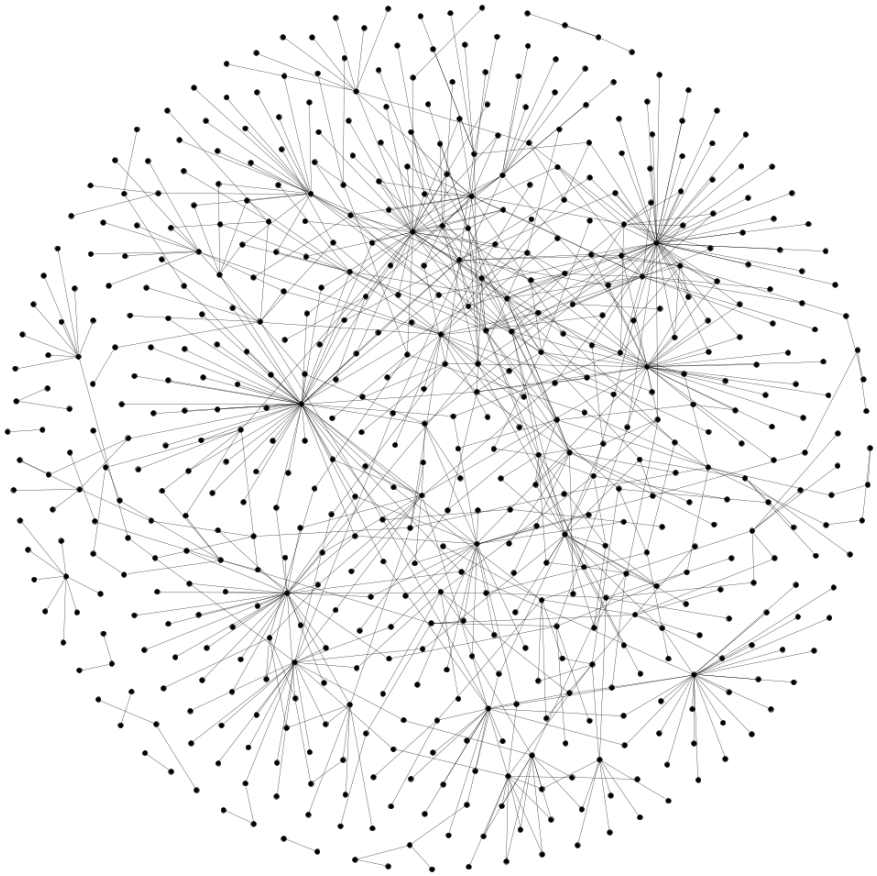


Figure 4.5: Function Call Graph of a Wooboo malware sample, rendered using Gephi with the Fruchterman Reingold layout.

4.3.3 Centrality Analysis

Now it's time to perform Centrality Analysis on the sensitive APIs present inside the extracted graphs. To do so, we use NetworkX [HSSC08], a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.

NetworkX

This Python package has built-in functions to compute the four centrality measures we are after:

- degree_centrality(G)

- katz_centrality(G, $alpha = 0.1$, $beta = 1.0$, $max\_iter = 1000$, $tol = 1e-06$, $nstart = None$, $normalized = True$, $weight = None$)

- closeness_centrality(G, $u = None$, $distance = None$, $wf\_improved = True$)

- harmonic_centrality(G, $nbunch = None$, $distance = None$, $sources = None$)

where G represents the input graph, *alpha* is an attenuation factor, *beta* is a weight attributed to the immediate neighborhood, max_iter is the maximum number of iterations, *tol*is the error tolerance used to check convergence, $wf\_improved$ makes the algorithm use the Wassermann and Faust improved formula [WF94].
Since Wu *et al.* haven't specified any parameters for the measures, we'll use the default settings for our tests. For the harmonic centrality normalization is also needed, since the NetworkX package doesn't perform it automatically.
By writing a simple Python script we computed the centrality measures of the sensitive APIs in our input graphs and generated and exported 5000 matrix (our images) using the Numpy package [HMvdW+20].

4.3.4 Contrastive Learning

To be able to replicate *IFDroid*, we first need to create the custom ResNet18 that supports grayscale 42*42 pixels images, and then we need to write the Supervised Contrastive Loss [KTW+20] function.
To achieve this we used the PyTorch Python package [PGM+19], an open-source deep learning framework that is widely used for developing and training artificial neural networks.

ResNet18

The first difference between a standard ResNet18 and our embedder is the number of channels of the image. To solve this, we pass $image\_channels$ as a variable, so that the first Convolutional Layer has the correct number of $in\_channels$.
The second difference is in the last layer: normally a ResNet18 ends with a Linear Layer that transforms the high-level features extracted by the convolutional layers into class scores. But since we are training an embedder, we don't need this final step and instead we normalize the 512 tensor generated by layer4.
The difference in size is already taken into account in the original code, thanks to the $AdaptiveAvgPool2d$ at the end of the model.

```
class ResNet_18(nn.Module):

    def __init__(self, image_channels):

        super(ResNet_18, self).__init__()
        self.in_channels = 64
        self.conv1 = nn.Conv2d(image_channels, 64, kernel_size=7,
                               stride=2, padding=3, device=device)
        self.bn1 = nn.BatchNorm2d(64, device=device)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2,
                                    padding=1)

        self.layer1 = self.__make_layer(64, 64, stride=1)
        self.layer2 = self.__make_layer(64, 128, stride=2)
        self.layer3 = self.__make_layer(128, 256, stride=2)
        self.layer4 = self.__make_layer(256, 512, stride=2)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))

    def __make_layer(self, in_channels, out_channels, stride):

        identity_downsample = None
        if stride != 1:
            identity_downsample = self.identity_downsample(in_channels,
                                                           out_channels)

        return nn.Sequential(
            Block(in_channels, out_channels,
                  identity_downsample=identity_downsample,
                  stride=stride),
            Block(out_channels, out_channels)
        )

    def forward(self, x):

        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = x.view(x.shape[0], -1)
        return nn.functional.normalize(x.squeeze(0), dim=0)

    def identity_downsample(self, in_channels, out_channels):

        return nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=2,
                      padding=0, device=device),
            nn.BatchNorm2d(out_channels, device=device)
        )
```

```
class Block(nn.Module):

    def __init__(self, in_channels, out_channels,
                 identity_downsample=None, stride=1):
        super(Block, self).__init__()
        self.block = nn.Sequential(nn.Conv2d(in_channels, out_channels,
                                   kernel_size=3, stride=stride,
                                   padding=1, device=device),
                                   nn.BatchNorm2d(out_channels,
                                   device=device),
                                   nn.Conv2d(out_channels, out_channels,
                                   kernel_size=3, stride=1, padding=1,
                                   device=device),
                                   nn.BatchNorm2d(out_channels,
                                   device=device)
                                  )
        self.relu = nn.ReLU()
        self.identity_downsample = identity_downsample

    def forward(self, x):
        identity = x
        x = self.block(x)
        if self.identity_downsample is not None:
            identity = self.identity_downsample(identity)
        x += identity
        x = self.relu(x)
        return x
```

Supervised Contrastive Loss

Now that we've written the code for the embedder, the next stop is to define the loss function. As previously stated, we are gonna employ the Supervised Contrastive Loss defined by Khosla *et al.* [KTW+20].

```
class SupConLoss(nn.Module):
    def __init__(self, temperature=0.07):
        super(SupConLoss, self).__init__()
        self.temp = temperature

    def forward(self, features, labels):
        loss = 0
        for i in range(0, len(labels)):
            denom_sum = 0
            for j in range(0, len(labels)):
                if i != j:
                    denom_sum += torch.exp(torch.dot(features[i],
                                                     features[j])/
                                                     self.temp)
            mid_sum = 0
            positives = 0
            for j in range(0, len(labels)):
                if i != j and labels[i] == labels[j]:
                    mid_sum += torch.log(torch.exp(torch.dot(features[i],
                                                             features[j])
                                                             /self.temp)
```

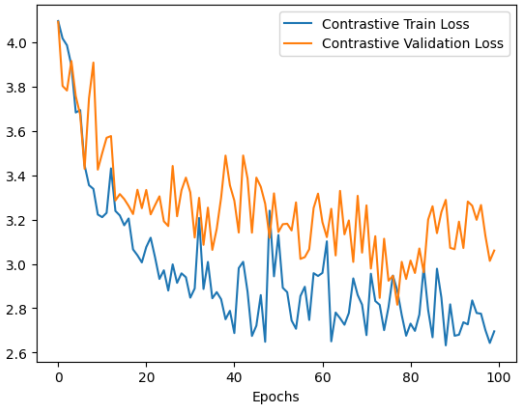
```
                                                                    /denom_sum)
                    positives += 1
            if positives>0:
                loss += mid_sum*(-1)/positives

    return loss/len(labels)
```

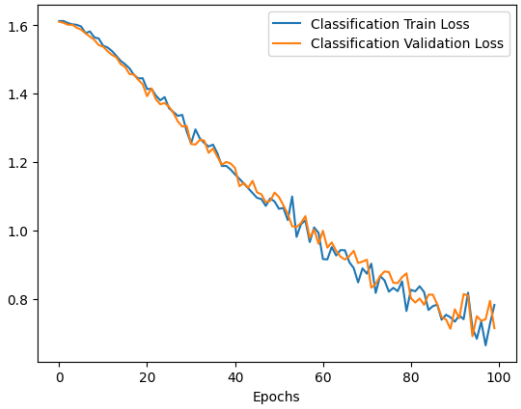
This is a simple PyTorch implementation of the Equation 4.5. The only difference being the normalization performed on the return line, so that the loss value doesn't depend on the batch size.

4.3.5 Results

It's now time to train our model. We used both a NVIDIA Tesla T4 (provided by Leonardo S.p.A.) and a NVIDIA GeForce RTX 3050 interchangeably for this experiment.
As indicated by Wu *et al.* [WDZ+22], we train both our encoder and classifier for 100 epochs. At epoch 0, the encoder has a train loss of 4.09590 and a validation loss of 4.09241, while at epoch 99 it has a train loss of 2.69575 and a validation loss of 3.06028. Our classifier, on the other hand, at epoch 0 has a train loss of 1.61223 and a validation loss of 1.61022, while at epoch 99 it has a train loss of 0.78163 and a validation loss of 0.71369.



(a) Encoder Loss



(b) Classifier Loss

Figure 4.6: Variation of the Loss over the epochs for the Android encoder and classifier.

To measure the effectiveness of our model, we leverage some widely used metrics (Table 4.2) to then compare them to *IFDroid* results.

Metrics	Abbr	Definition
True Positive	**TP**	#malware in family f are correctly classified into family f
True Negative	**TN**	#malware not in family f are correctly not classified into family f
False Positive	**FP**	#malware not in family f are incorrectly classified into family f
False Negative	**FN**	#malware in family f are incorrectly not classified into family f
True Positive Rate	**TPR**	$TP/(TP+FN)$
False Negative Rate	**FNR**	$FN/(TP+FN)$
True Negative Rate	**TNR**	$TN/(TN+FP)$
False Positive Rate	**FPR**	$FP/(TN+FP)$
Precision	**P**	$TP/(TP+FP)$
Recall	**R**	$TP/(TP+FN)$
F-measure	**F1**	$2*P*R/(P+R)$
Classification Accuracy	**CA**	$(TP+TN)/(TP+TN+FP+FN)$

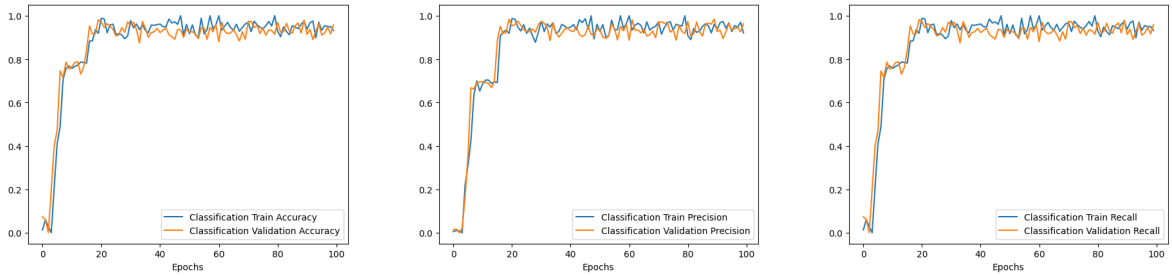
Table 4.2: List of the various metrics used in this experiment.

Wu *et al.* only provided the F-measure of *IFDroid* in their paper [WDZ+22]. We were able to find, in an older version of the same paper [WDZ+21], the True Positive Rate and False Positive Rate of the malware classification, but it must be kept in mind that represents an older build of *IFDroid*, and it doesn't accurately represent its full capabilities. In Table 4.3 the F1Scores are from the newer version, while the TPR and FPR are from the older one.

Family	**TPR(%)**		**FPR(%)**		**F1**	
	IFDroid	*ours*	*IFDroid*	*ours*	*IFDroid*	*ours*
airpush	85.5	100	0.1	2.5	92.5	95.2
droidkungfu	98	90.0	0	0.6	99.5	93.5
feiwo	?	92.5	?	0.6	94.4	94.9
utchi	100	100	0	0	100	100
wooboo	?	92.5	?	2.5	98.2	91.4

Table 4.3: Result metrics of both the original and our tests of *IFDroid*.

Last but not least, we show the other metrics improvements of our model over the epochs.



(a) Accuracy **(b)** Precision **(c)** Recall

Figure 4.7: Variation of different effectiveness metrics over the epochs for the Android classifier.

4.4 Conclusions

Even with a smaller dataset, we were able to witness the capabilities of *IFDroid*. It must be stated that the classifier was still trainable for many more epochs without risk of overfitting, but we decided to use the exact same variables as the original paper for the sake of this Thesis.
Regardless, the ideas shown in this Chapter are much more general than it could seem. Training an encoder using Supervised Contrastive Learning can be used with any kind of input, not only images. Many common classification methods may benefit from contrastive learning, such as:

- **Binary-based methods**: Executable files, or binaries, are binary files containing machine code for the computer to execute. The binary carries all necessary information that enables the program's execution in the target environment. Most of the works using this method take in byte sequences as input features for the learning models, resulting in very fast feature extraction but also making them susceptible to simple obfuscations techniques.

- **Opcode-based methods**: An opcode, abbreviated from operation code, is the portion of a machine language instruction specifying the operation to be performed. Opcode sequences, as the output of many reverse engineering tools, carry fine-grained information about the execution logic of the program. Due to the extremely fine granularity of opcode, a graph representation yielded by opcode may show a size rendering fast analysis impossible. Integrating opcode as function calls (blocks of opcodes that realize particular functionalities) can not only significantly reduce the size for graph representation but also result in improved modeling of the malware behavior at a higher level.

- **Graph-based methods**: The malicious behavior of malware can be characterized by certain components in some representative structures obtained from malware binaries that reflect the execution logic of the program. In the literature, Control Flow Graphs (CFGs) and Function Call Graphs (FCGs) are among the most widely adopted representative structures to serve this purpose. By extracting the vector representations of the semantics in the graph through graph embedding, we can capture essential discriminating information from malware samples.

- **API-based methods**: Application programming interface (API), a software interface that offers a service to other pieces of software, is used in high-level programming to invoke system calls at the low level. as far as dynamic analyses are concerned, API sequences captured during the program's execution time are reported as the most significant data type that facilitates malware analysis.

Many other types of information can be obtained from static analysis and serve as behavioural indicators of malware.

Chapter 5

Linux IoT Malware Classification

Now that we verified the replicability of the model made by Wu *et al.*, we ask ourselves the question: could the same ideas be applied to the classification of Linux IoT?
Before delving into the experimenting, we need to properly translate the steps performed in Chapter 4 to the Linux IoT environment. But even earlier than that, we need a dataset.

5.1 Motivation

With the rise in popularity of Internet of Things (IoT) devices, while our digital life has been made easier, our cybersecurity has been exposed to new threats, creating unprecedented challenges in safeguarding our personal information and ensuring the integrity of interconnected systems.
In particular, due to the open-sourceness of malware programs' code such as Mirai [AAB+17], a botnet that took the Internet by storm in late 2016 when it overwhelmed several high-profile targets with massive distributed denial-of-service (DDoS) attacks, numerous malware variants have appeared, rendering the protection of IoT devices more challenging while also making it harder to disambiguate the ownership, lineage, and correct label of these malware variants.
Contrastive Learning is a technique that hasn't been employed many times in the IoT environment. To the best of our knowledge, only one paper uses (Self-supervised) Contrastive Learning for the classification of IoT Malware [DTBH+22]. That said, its outstanding results in other fields make us confident about its capabilities.

5.2 CUBE-MALIOT-2021 Dataset

Since IoT security hasn't been object of research for a long time, availability of datasets has been very scarce. However, it has now become more and more important to explore the cybersecurity of IoT devices, especially because they have been employed in many sensitive domains, including healthcare, transportation and agriculture.
After thorough research, we were able to gain access to a dataset, provided by CrySyS Lab and Ukatemi Technologies, called "CrySyS-Ukatemi BEnchmark: MALware for IOT devices 2021", or CUBE-MALIOT-2021[1] for short. The dataset consists of 29,209 ELF binaries compiled for the ARM platform and 18,715 ELF binaries compiled for the MIPS platform.
All metadata of the raw binaries is available as JSON documents, where the file name of each

[1] https://github.com/CrySyS/cube-maliot-2021

JSON document is the SHA256 hash of the corresponding malicious binary. These JSON documents contain information obtained from the binaries (e.g., ELF header information, file size, etc.) and information obtained from the VirusTotal[2] analysis reports of the samples (e.g., first submission date to VirusTotal, antivirus labels assigned by antivirus products, etc.). Accurate antivirus labels were computed with a slightly modified version of AVClass (version 1) [SRKC16].


```
{
    "architecture": "ARM",
    "av_labels_generation_date": "2020-08-08T02:18:53",
    "avclass_labels": {
        "ddos": 3,
        "gafgyt": 14,
        "mirai": 2
    },
    "elf_header_info": {
        "abi_version": 0,
        "class": "ELF32",
        "data": "2's complement, little endian",
        "entrypoint": 33200,
        "hdr_version": "1 (current)",
        "num_prog_headers": 4,
        "num_section_headers": 24,
        "obj_version": "0x1",
        "os_abi": "UNIX - System V",
        "type": "EXEC (Executable file)"
    },
    "first_submission_date": "2017-03-11T17:23:29",
    "magic": "ELF 32-bit LSB executable, ARM, version 1
              (SYSV), statically linked, not stripped",
    "magic_is_stripped": false,
    "magic_linking": "static",
    "md5": "356e84a502f1acd1f6363d17ce544312",
    "sha1": "3547c0c8030e93274fcf25c3701df0e8634ce682",
    "sha256": "0bbe52f239f903b9e922a795163e417dc1e1a3835b46
               22656bff46320adb0ef2",
    "size": 134804,
    "tlsh": "91D30A05D8509737C2D22BBAF7AA42CE73261FA8579733
             1296287EF41FE1B9D1D39120"
}
```


Code Listing 5.1: Metadata of a Gafgyt ARM malware sample.

[2]For more information, visit www.virustotal.com

In the Code Listing 5.1 is shown the JSON file of an ARM malware sample. To properly divide our dataset between malware families, we write a simple python script that puts each malware in the family that has the highest AVClass score in its JSON. Following this procedure, the malware described in Code Listing 5.1 would be classified as a Gafgyt malware.

5.3 Static Analysis for ELF files

In the previous chapter we worked with APK (Android Package) files, a format used to distribute and install applications on the Android operating system. Now that we've moved to the Linux Environment we'll be working with ELF (Executable and Linkable Format) files, a format used for executables on Unix and Unix-like operating systems.
Because of this, we need a new way to extract the Function Call Graph from the malware samples. To do so, we will use Radare2, an open-source framework for reverse engineering and binary analysis.
The analysis of a malware would proceed as follows:

```
C:\Users\Sandman\Desktop>r2 0013b477ce0de71e48076e36217baf4f3dc5e4a325210d17320a178a41a584d4
[0x00008194]> aaa
INFO: Analyze all flags starting with sym. and entry0 (aa)
WARN: Cannot find basic block for switch case at 0x000106a4 bbdelta = 12
INFO: Analyze all functions arguments/locals (afva@@@F)
INFO: Analyze function calls (aac)
INFO: Analyze len bytes of instructions for references (aar)
INFO: Finding and parsing C++ vtables (avrr)
INFO: Finding xrefs in noncode section (e anal.in=io.maps.x)
INFO: Analyze value pointers (aav)
INFO: aav: 0x00008000-0x0001aee8 in 0x8000-0x1aee8
INFO: Emulate functions to find computed references (aaef)
INFO: Type matching analysis for all functions (aaft)
INFO: Propagate noreturn information (aanr)
INFO: Integrate dwarf function information
INFO: Use -AA or aaaa to perform additional experimental analysis
[0x00008194]> afl
0x00008194    1     44 entry0
0x000080f0    3     48 sym.__do_global_dtors_aux
0x00008134    1     32 sym.frame_dummy
0x0000d6bc    1     16 sym.anti_gdb_entry
0x0000d6d4    1     64 sym.resolve_cnc_addr
0x000113dc    4    152 sym.table_unlock_val
0x00012a54    1     40 sym.inet_addr
0x000188cc   17    240 sym.inet_aton
0x00011318    1     32 sym.table_retrieve_val
0x0001133c    4    152 sym.table_lock_val
0x0000e1dc    3    196 sym.setup_connection
0x00015760    3    100 sym.__GI_close
0x00015a08   16    212 sym.__libc_enable_asynccancel
0x00015980   11    136 sym.__libc_disable_asynccancel
0x00012f98    3     60 sym.socket
0x00016190    1     16 sym.__aeabi_read_tp
0x000119b0    3     36 sym.util_zero
0x00012038    9    228 sym.__libc_fcntl
0x00012b04    3    108 sym.connect
0x00012ac0    3     60 sym.__sys_connect
0x0000e2a4    7    340 sym.add_auth_entry
0x000122dc    3     64 sym.__syscall_select
0x000125f0    5    200 sym.fd_to_DIR
0x00012c44    3     60 sym.__sys_recv
0x00012cf8    3     68 sym.__sys_recvfrom
0x00012dc8    3     60 sym.__sys_send
0x00012e7c    3     68 sym.__sys_sendto
...
```

The `r2` command launches the Radare2 interactive command-line interface (CLI) and opens the specified binary file for analysis. We then use the `aaa` command to trigger a series of automatic analyses to gather information about the binary, including identifying functions, creating basic blocks, and recognizing data structures. We can use the `afl` command to list all the functions in the analyzed binary to have a better understanding of its behaviour. A subset of these functions will take the place of the API calls in the previous chapter image generation.
This framework also gives us access to the generation of multiple graphs:

```
[0x00008194]> ag?
Usage: ag<graphtype><format> [addr]
Graph commands:
| aga[format]                 data references graph
| agA[format]                 global data references graph
| agc[format]                 function callgraph
| agC[format]                 global callgraph
| agd[format] [fcn addr]      diff graph
| agf[format]                 basic blocks function graph
| agi[format]                 imports graph
| agr[format]                 references graph
| agR[format]                 global references graph
| agx[format]                 cross references graph
| agg[format]                 custom graph
| agt[format]                 tree map graph
| ag-                         clear the custom graph
| agn[?] title body           add a node to the custom graph
| age[?] title1 title2        add an edge to the custom graph

Output formats:
| <blank>                     ascii art
| *                           r2 commands
| b                           braile art rendering (agfb)
| d                           graphviz dot
| g                           graph Modelling Language (gml)
| j                           json ('J' for formatted disassembly)
| k                           sdb key-value
| m                           mermaid
| t                           tiny ascii art
| v                           interactive ascii art
| w [path]                    write to path or display graph image
```

The one that we need is the "global callgraph". After a few tests, we landed on graphviz dot as the best output format. Using the command `agCdw` we save the .gv file of the global callgraph.
Using r2pipe, a set of libraries that provide a way to interact with Radare2 through various programming languages, we can automate the creation of these graphs for all of our malware samples with a simple python script.

5.4 Sensitive Functions List

When creating a Global callgraph with Radare2, the nodes of the graph will have either "fcn." or "sym." as prefix. The "fcn." nodes represent UDFs (User Defined Functions), while "sym." modes represent Library Functions declared in the header of the binary. In graph form, since we are studying the behaviour of the malware based on the nodes names, UDFs don't give us any information, so our only focus will be Library functions.

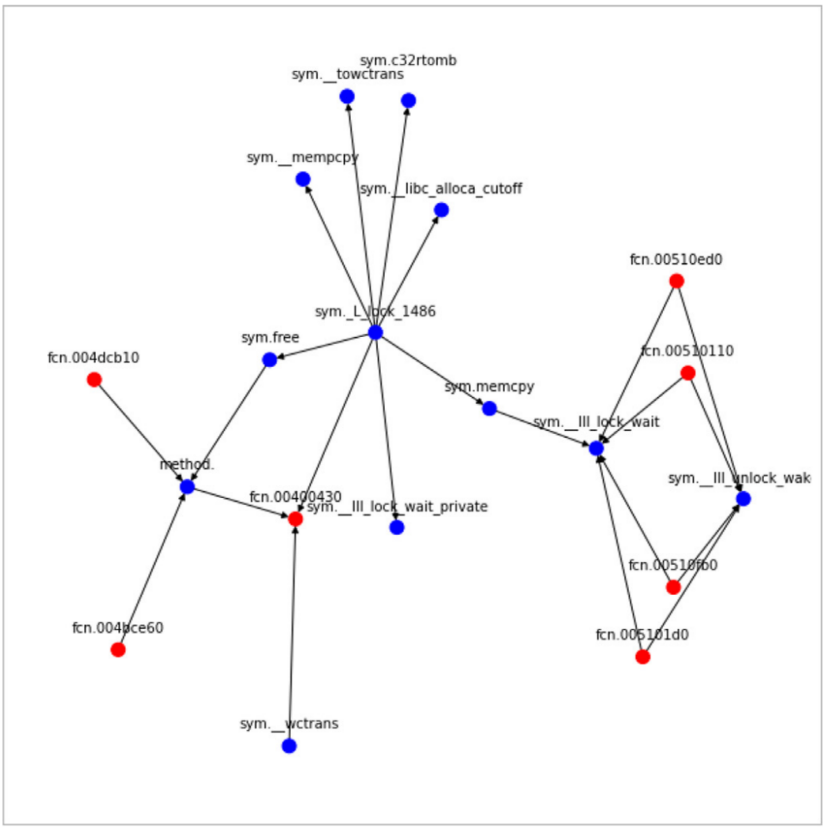


Figure 5.1: Example of a Function Call Graph extracted using Radare2. The red nodes are UDFs, while the blue nodes are Library functions.

Our objective now is to find a set of Library functions that are able to correctly describe the malicious behaviour of a program. Differently from Android, there haven't been many studies on the most used libraries by malware families. That said, we did find a study [AO21] analyzing the most common toolchains used to build Linux IoT malware (Table 5.1), *i.e.* a set of tools that compiles source code into executables that can run on a target device. It includes a compiler (a tool that translates high-level source code into machine code), a linker (a program that takes one or more object files (generated by a compiler or an assembler) and combines them into a single executable file), and run-time libraries (collections of precompiled routines, functions, and procedures).
In their research, they found out that most Linux malware are built with toolchains using well-known toolchain building tools available in binary form online, rather than customized toolchains for anti-analysis.

Building tool: toolchain components	Number of samples
Firmware Linux 0.9.6 : GCC 4.1.2, binutils 2.17, uClibc 0.9.30.1	3,830
Aboriginal Linux 1.1.0 : GCC 4.2.1, binutils 2.17, uClibc 0.9.32	8
Aboriginal Linux 1.1.1 : GCC 4.2.1, binutils 2.17, uClibc 0.9.32.1	6
Aboriginal Linux 1.2.0 : GCC 4.2.1, binutils 2.17, uClibc 0.9.33.2	17
Aboriginal Linux 1.2.1 : GCC 4.2.1, binutils 397a64b3, uClibc 0.9.33.2	11
Aboriginal Linux 1.2.4 : GCC 4.2.1, binutils 397a64b3, uClibc 0.9.33.2	27
Aboriginal Linux 1.2.6 : GCC 4.2.1, binutils 2.17, uClibc 0.9.33.2	22
Aboriginal Linux 1.4.3 : GCC 4.2.1, binutils 397a64b3, uClibc 0.9.33.2	6
Aboriginal Linux 1.4.4 : GCC 4.2.1, binutils 2.17, musl 1.1.12	36
Buildroot 2018.08 (Bootlin toolchain) : GCC 7.3.0, binutils 2.29.1, uClibc-ng 1.0.30	22
Buildroot 2018.08 (Bootlin toolchain) : GCC 7.3.0, binutils 2.29.1, musl 1.1.19	2
Buildroot 2018.08 (Synopsys toolchain) : GCC 7.1.1, binutils2.29, uClibc-ng 1.0.26	2
Crosstool-NG 1.24.0-rc1: GCC 8.2.0, binutils 2.30, musl 1.1.19	2

Table 5.1: Result of toolchain identification performed by Akabane *et al.* [AO21].

By using these findings, we can focus on the "run-time libraries" portion of the toolchains to be able to list the most common library functions found in Linux IoT malware. As shown in Table 5.2, the most used C Library in Linux IoT malware is uClibc, a small and lightweight C library designed for embedded systems and systems with limited resources.

C library	Number of samples	Percentage
uClibc	3,951	99.00%
musl	40	1.00%

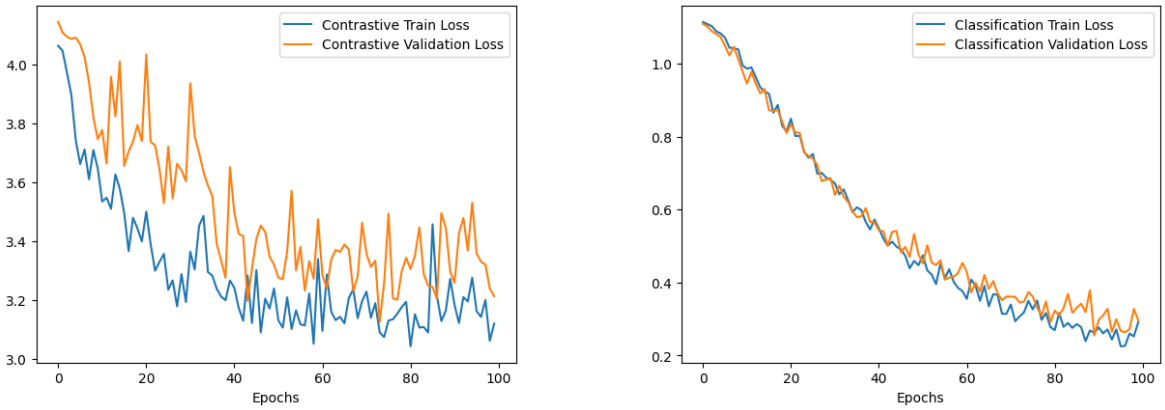
Table 5.2: Most used C libraries by malware samples as found by Akabane *et al.* [AO21].

We decided to focus on the functions in the uClibc 0.9.30.1, since it's the version contained in the most commonly used toolchain (Firmware Linux 0.9.66, which is described in the Mirai installation guide [As16]). This way, we restricted ourselves to 2214 library functions, which will constitute our "sensitive" functions list.
Unfortunately, a problem arises due to the fact that many IoT malware samples includes static linking of library functions, and their symbols such as function names and addresses are stripped, hindering function-level analysis. A way to solve this problem has been proposed by Akabane and Okamoto in "Identification of library functions statically linked to Linux malware without symbols" [AO20] and will be discussed later in this chapter. For the sake of this thesis, we'll work on the unstripped malware samples present in the CUBE-MALIOT-2021 Dataset.

5.5 Results

It's now time to train our new model. This time, we will generate 95*95 pixel grayscale image (with a 169 pixels padding, since $95*95-2214*4=169$) following the same criteria as *IFDroid*. We used once again both a NVIDIA Tesla T4 (provided by Leonardo S.p.A.) and a NVIDIA GeForce RTX 3050 interchangeably for this experiment. We also use the same Parameters for both the encoder and the classifier as *IFDroid*.
We reduced the dataset to the 3 malware families with the most unstripped samples: Gafgyt,

Mirai and Ddostf. These malware families target and compromise IoT devices, turning them into a botnet used for large-scale distributed denial-of-service (DDoS) attacks. In total we'll use 1000 Gafgyt samples, 1000 Mirai samples and 707 Ddostf samples.
We train our model for 100 epochs both both our encoder and our classifier. At epoch 0, the encoder has a train loss of 4.06207 and a validation loss of 4.14268, while at epoch 99 it has a train loss of 3.12014 and a validation loss of 3.21261. Our classifier, on the other hand, at epoch 0 has a train loss of 1.11393 and a validation loss of 1.10900, while at epoch 99 it has a train loss of 0.29114 and a validation loss of 0.29571.



(a) Encoder Loss **(b)** Classifier Loss

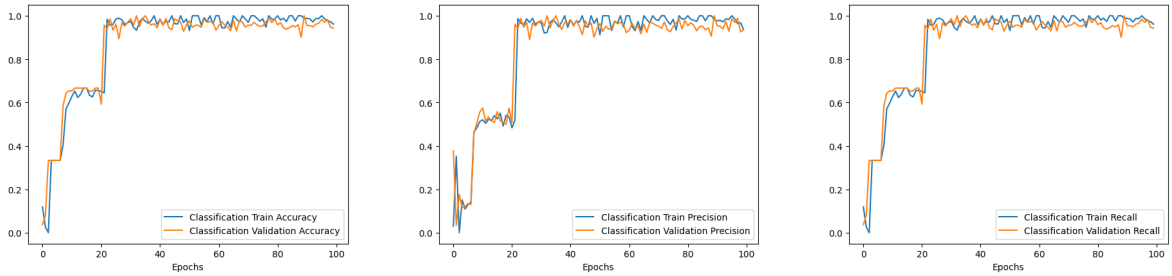
Figure 5.2: Variation of the Loss over the epochs for the IoT encoder and classifier.

In Table 5.3 we show some widely used metrics (Table 4.2) to measure the effectiveness of our model.

Family	TPR(%)	FPR(%)	F1Score	Accuracy	Precision	Recall
gafgyt	100	0.0	100	100	100	100
mirai	87.5	0.0	93.3	87.5	100	87.5
ddostf	100	6.2	91.8	100	84.8	100

Table 5.3: Result metrics of our IoT malware classifier.

Last but not least, we show some metrics improvements of our model over the epochs.



(a) Accuracy **(b)** Precision **(c)** Recall

Figure 5.3: Variation of different effectiveness metrics over the epochs for the IoT classifier.

5.6 Conclusions

Even while working with only a small amount of samples, the neural network was able to achieve great results in the classification of these malware families. That being said, many compromises where made to be able to reach those results, especially focusing only on unstripped samples.
There exists a few different ways to identify a stripped function:

- **IDA F.L.I.R.T** [HR15]: checks, at each byte of the program being disassembled, whether it can mark the start of a standard library function. Each function is represented by a pattern, the first 32 bytes of a function where all variant bytes (*i.e.*, bytes that don't have any relevance in the identification of the function) have a special character, and the CRC16 (cyclic redundancy check) of the bytes starting from position 33 until the first variant byte. However, since modern real-world libraries contain several functions starting with the same bytes and the checksum is only computed until the first variant byte appears, unique matching of C library functions isn't guaranteed to be achieved.

- **FCatalog** [xor15]: identifies a function by comparing its instruction sequences to a database. It can identify a function even if the order of instructions differ or an instruction is replaced with an alternative. However, they may falsely identify different functions as being the same if they have the same sequence of instructions but different registers.

- **BinDiff** [DR05]: identifies a function by comparing control flow graphs of two functions. They are tolerant to instruction replacement and sequence reordering in a node of the control flow graph, but they may falsely identify different functions as being the same if they have the same control flow graph.

- **BinSequence** [HYD17]: combines instruction sequence similarity and control flow graph similarity to identify functions. May also falsely identify different functions as being the same if a pair of functions have a similar sequence of instructions or similar control flow graphs.

Function identification by pattern matching is difficult because compiler-generated machine code differs for each toolchain. Moreover, function identification based on the similarity of machine code and/or its control flow graph gives false results if two different functions have the same sequence of instructions but different registers or different immediate data. In fact, many C library functions result in false positives for this reason.
Due of these problems, Akabane and Okamoto conducted some accurate studies [AO20][AO21] focusing on library functions recognition for Linux IoT malware.
Because of the high diversity of C libraries for desktop computers and servers, pattern matching for identification of these library functions requires generation of various patterns from a large number of libraries. This requires a lot of time and effort. In contrast, there are few C libraries for embedded devices, and fewer versions. Furthermore, their study has confirmed that malware developers prefer toolchains built with Firmware Linux series [Lan02].
Their results showed that pattern matching identified toolchains used to build 97.7% of 2,256 samples with the Intel 80386 architecture, indicating that pattern matching is indeed effective for function identification of Linux malware on embedded devices.

Chapter 6

Future Works

In this chapter we will present some ideas on how to expand on the work presented in this thesis, both on the Deep Learning and the Malware Analysis aspects.

6.1 Generalized Supervised Contrastive Loss

As can be seen in Code Listing 5.1, Anti-Viruses (AVs) don't always agree on the label provided for a specific malware sample. However, since supervised contrastive learning (SupCon [KTW+20]) inherently employs label information in a binary form, during our experiments we decided to use as a ground-truth label the argmax of the possible labels.
A new approach could be to represent label information as a probability distribution. For example, using the AVclass labels in Code Listing 5.1:

```
"avclass_labels": {
    "ddos": 3,
    "gafgyt": 14,
    "mirai": 2
}
```

the label vector for SupCon would be $[0, 1, 0]$, while, with the new approach, the new label vector would be $\left[\frac{3}{19}, \frac{14}{19}, \frac{2}{19}\right] \approx [0.16, 0.74, 0.10]$.
To properly take advantage of this new label representation, a new loss function is needed: the Generalized Supervised Contrastive Loss (GenSCL [KCP22]).
Proposed by Kim *et al.*, this work builds on the supervised contrastive loss but is capable of leveraging label information as a probability distribution.
To fully utilize the label information in the form of the probability distribution, generalized supervised contrastive aims to minimize cross-entropy between label similarity space $Y(i) \equiv \{\text{sim}(y_i, y_l)\}_{l \in A(i)}$ and latent similarity space $Z(i) \equiv \{P_{il}\}_{l \in A(i)}$ with respect to anchor i, where

$$\text{sim}(u, v) = \frac{u^T v}{\|u\|\|v\|} \quad \text{and} \quad P_{ij} = \frac{\exp\left(\frac{z_i \cdot z_j}{\tau}\right)}{\sum_{a \in A(i)} \exp\left(\frac{z_i \cdot z_a}{\tau}\right)} \tag{6.1}$$

Being I the set of indices of the inputs, z_i the embedding of the input x_i, y_i the label of the input x_i, $\tau \in \mathbb{R}^+$ a scalar temperature parameter, and $A(i) \equiv I \backslash \{i\}$, the generalized supervised contrastive loss takes the following form:

$$\begin{aligned}\mathcal{L}^{gen} = \sum_{i \in I} \mathcal{L}_i^{gen} &= \sum_{i \in I} \frac{1}{|A(i)|} \mathrm{CE}(Y(i), Z(i)) \\ &= \sum_{i \in I} \frac{-1}{|A(i)|} \sum_{j \in A(i)} \mathrm{sim}\left(y_i, y_j\right) \log \frac{\exp\left(\frac{z_i \cdot z_j}{\tau}\right)}{\sum_{a \in A(i)} \exp\left(\frac{z_i \cdot z_a}{\tau}\right)}\end{aligned} \tag{6.2}$$

Here CE denotes cross-entropy loss. As shown in Eq. 6.2, generalized supervised contrastive loss no longer depends on the positive contrasts set $P(i)$ which enforces the use of one-hot encoded labels, as shown in Eq. 4.5. Intuitively, the generalized supervised contrastive loss considers drawing positive contrasts to an anchor delicately according to label similarity, instead of pulling all positive contrasts evenly as in previous contrastive losses (Figure 6.1).

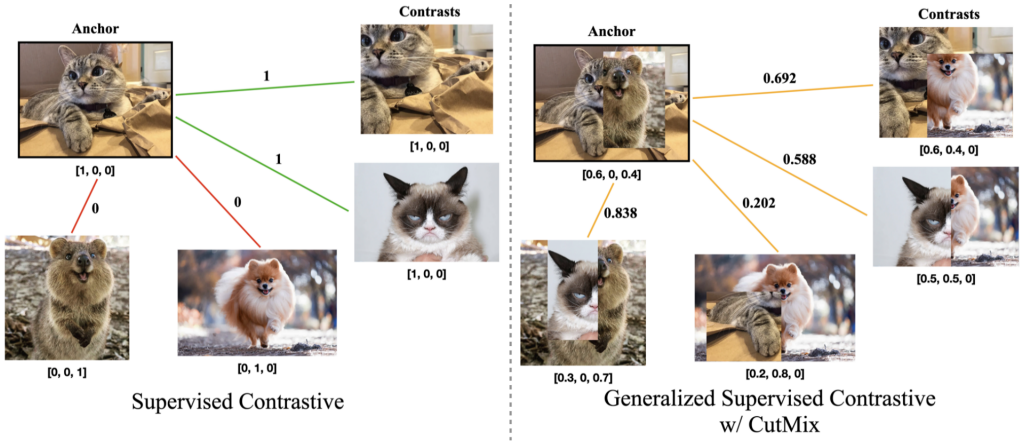


Figure 6.1: Generalized Supervised vs. Supervised Contrastive Losses.

Using this new kind of approach wouldn't probably increase performance per se, but it will create embeddings capable to better represent the malware sample from which they were generated.

6.2 Symbol Extraction and Recognition

Most Linux IoT malware classifiers and clustering rely on symbols to be able to extract information from the samples. An extensive study on Linux IoT malware has been provided by Emanuele Cozzi [CVD+20]. In it, he shows that, as of 2020, over 90% of Linux IoT malware samples were statically linked, and half of them were stripped, a tendency that has more that likely increased over the years.
Another tendency that hinders malware classification is code reuse. Since many of the most common malware families have their source code freely available online, many other families simply copy and paste their function in their own code. It can be seen in Figure 6.2 how many functions are in common between the top-10 malware families, as reported by Cozzi [CVD+20].

VS	Gafgyt	Mirai	Tsunami	Dnsamp	Hajime	Ddostf	Lightaidra	Pnscan	Skeeyah	VPNFilter
Gafgyt		115	189	3	1	2	18	-	-	-
Mirai			63	1	1	-	2	-	-	-
Tsunami				4	-	3	1	-	-	-
Dnsamp					-	65	-	-	-	-
Hajime						-	-	-	-	-
Ddostf							-	-	-	-
Lightaidra								-	-	-
Pnscan									-	-
Skeeyayh										-
VPNFilter										

Figure 6.2: Common functions across top-10 malware families.

To be able to divide library function from user defined function in stripped malware, Cozzi built a database of symbols extracted from different versions of Glibc and uClibc. He also used the non-stripped binaries to create a database of user defined functions used in IoT malware, and then used Diaphora[1], the most advanced program diffing (*i.e.*, comparing and highlighting the differences between two sets of data) tool, to check if the same functions were used in the stripped binaries.
A similar idea may be employed also in our case: a database with sensitive functions (or variants of them) could be created, and then we could analyze the centrality measures of the functions contained in the sample's call graph that, using Diaphora, obtained a high enough similarity score to one of the function inside the database. These function could be both library functions or specific used defined function that are commonly found in malware families. However, an extensive study on what these functions could be is needed before being able to test this idea.

6.3 Final Considerations

Even though malware analysis for IoT is still in its early stages, the threat is growing rapidly. Due to the lack of security of most embedded devices, most malicious programs don't even need to be obscured or packed like most of their Windows counterparts. However it won't take long before proper countermeasures to debugging and analysis are applied also to IoT malware.
Many more advanced studies, like Cozzi's [Coz20], are needed to be able to fully understand how they work, but, even more important than that, how they evolve. Especially because, nowadays, embedded devices are becoming a bigger and bigger part of our lives, while also often collecting and transmitting sensitive data.

[1]For more information, visit www.diaphora.re

Bibliography

[AAB+17] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al., *Understanding the mirai botnet*, 26th USENIX security symposium (USENIX Security 17), 2017, pp. 1093–1110.

[ABKLT16] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon, *AndroZoo: Collecting millions of android apps for the research community*, Proceedings of the 13th international conference on mining software repositories, MSR '16, 2016, pp. 468–471.

[ABKLT24] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon, *AndroZoo*, `https://androzoo.uni.lu/`, 2024, last accessed: 2024-02-07.

[And18] Dennis Andriesse, *Practical binary analysis: build your own Linux tools for binary instrumentation, analysis, and disassembly*, no starch press, 2018.

[AO20] Shu Akabane and Takeshi Okamoto, *Identification of library functions statically linked to Linux malware without symbols*, Procedia Computer Science **176** (2020), 3436–3445.

[AO21] ———, *Identification of toolchains used to build IoT malware with statically linked libraries*, Procedia Computer Science **192** (2021), 5130–5138.

[AQP+22] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck, *Dos and Don'ts of Machine Learning in Computer Security*, 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 3971–3988.

[As16] Anna-senpai, *World's largest net: Mirai botnet, client, echo loader, CNC source code release*, `https://github.com/jgamblin/Mirai-Source-Code/blob/master/ForumPost.md`, 2016, last accessed: 2024-02-16.

[AZHL12] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie, *PScout: analyzing the android permission specification*, Proceedings of the 2012 ACM conference on Computer and communications security, 2012, pp. 217–228.

[BBD+16] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber, *On demystifying the android application framework: Re-Visiting android permission specification analysis*, 25th USENIX security symposium (USENIX security 16), 2016, pp. 1101–1118.

[BHJ09] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy, *Gephi: an open source software for exploring and manipulating networks*, Proceedings of the international AAAI conference on web and social media, vol. 3, 2009, pp. 361–362.

[Bis06] Christopher Bishop, *Pattern recognition and machine learning*, Springer, 2006.

[CGFB18] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti, *Understanding linux malware*, 2018 IEEE symposium on security and privacy (SP), IEEE, 2018, pp. 161–175.

[Coz20] Emanuele Cozzi, *Binary Analysis for Linux and IoT Malware*, Ph.D. thesis, Sorbonne université, 2020.

[CVD+20] Emanuele Cozzi, Pierre-Antoine Vervier, Matteo Dell'Amico, Yun Shen, Leyla Bilge, and Davide Balzarotti, *The Tangled Genealogy of IoT Malware*, Annual Computer Security Applications Conference, 2020, pp. 1–16.

[Des11] Anthony Desnos, *Androguard*, `https://github.com/androguard/androguard`, 2011, last accessed: 2024-01-31.

[Dir81] Paul Adrien Maurice Dirac, *The principles of quantum mechanics*, no. 27, Oxford university press, 1981.

[DLD+18] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang, *Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild*, Security and Privacy in Communication Networks: 14th International Conference, SecureComm 2018, Singapore, Singapore, August 8-10, 2018, Proceedings, Part I, Springer, 2018, pp. 172–192.

[DR05] Thomas Dullien and Rolf Rolles, *Graph-based comparison of executable objects (english version)*, Sstic **5** (2005), no. 1, 3.

[DTBH+22] Mirabelle Dib, Sadegh Torabi, Elias Bou-Harb, Nizar Bouguila, and Chadi Assi, *EVOLIoT: A self-supervised contrastive learning framework for detecting and characterizing evolving IoT malware variants*, Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security, 2022, pp. 452–466.

[EB22] Ghassan El Baltaji, *Anomaly Detection at the Edge implementing Machine Learning Techniques*, Master's thesis, Politecnico di Torino - Department of Mathematical Sciences (DISMA), March 2022.

[F+02] Linton C Freeman et al., *Centrality in social networks: Conceptual clarification*, Social network: critical concepts in sociology. Londres: Routledge **1** (2002), 238–263.

[FLL+18] Ming Fan, Jun Liu, Xiapu Luo, Kai Chen, Zhenzhou Tian, Qinghua Zheng, and Ting Liu, *Android malware familial classification and representative sample selection via frequent subgraph analysis*, IEEE Transactions on Information Forensics and Security **13** (2018), no. 8, 1890–1905.

[FR91] Thomas MJ Fruchterman and Edward M Reingold, *Graph drawing by force-directed placement*, Software: Practice and experience **21** (1991), no. 11, 1129–1164.

[GLQ+20] Liangyi Gong, Zhenhua Li, Feng Qian, Zifan Zhang, Qi Alfred Chen, Zhiyun Qian, Hao Lin, and Yunhao Liu, *Experiences of landing machine learning onto market-scale mobile malware detection*, Proceedings of the Fifteenth European Conference on Computer Systems, 2020, pp. 1–14.

[GSG19] Mohit Goyal, Ipsit Sahoo, and G Geethakumari, *HTTP Botnet Detection in IoT Devices using Network Traffic Analysis*, 2019 International Conference on Recent Advances in Energy-efficient Computing and Communication (ICRAECC), IEEE, 2019, pp. 1–6.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant, *Array programming with NumPy*, Nature **585** (2020), no. 7825, 357–362.

[HR15] Hex-Rays, *IDA F.L.I.R.T. technology: in-depth*, `https://hex-rays.com/products/ida/tech/flirt/in_depth/`, 2015, last accessed: 2024-02-16.

[HSSC08] Aric Hagberg, Pieter Swart, and Daniel S Chult, *Exploring network structure, dynamics, and function using NetworkX*, Tech. report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[HSTD+17] Médéric Hurier, Guillermo Suarez-Tangil, Santanu Kumar Dash, Tegawendé F Bissyandé, Yves Le Traon, Jacques Klein, and Lorenzo Cavallaro, *Euphony: Harmonious unification of cacophonous anti-virus vendor labels for android malware*, 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), IEEE, 2017, pp. 425–435.

[HYD17] He Huang, Amr M Youssef, and Mourad Debbabi, *Binsequence: Fast, accurate and scalable binary code reuse detection*, Proceedings of the 2017 ACM on Asia conference on computer and communications security, 2017, pp. 155–166.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, *Deep residual learning for image recognition*, Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.

[IS15] Sergey Ioffe and Christian Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, International conference on machine learning, pmlr, 2015, pp. 448–456.

[Kat53] Leo Katz, *A new status index derived from sociometric analysis*, Psychometrika **18** (1953), no. 1, 39–43.

[KCP22] Jaewon Kim, Jooyoung Chang, and Sang Min Park, *A Generalized Supervised Contrastive Learning Framework*, arXiv preprint arXiv:2206.00384 (2022).

[Kiv23] Anton Kivva, *IT threat evolution in Q3 2023. Mobile statistics*, `https://securelist.com/it-threat-evolution-q3-2023-mobile-statistics/111224/`, 2023, last accessed: 2024-01-22.

[KTW+20] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan, *Supervised Contrastive Learning*, Advances in neural information processing systems **33** (2020), 18661–18673.

[Lan02] Rob Landley, *Firmware Linux*, `https://landley.net/code/firmware/old`, 2002, last accessed: 2024-02-16.

[Mac03] David JC MacKay, *Information theory, inference and learning algorithms*, Cambridge university press, 2003.

[ML00] Massimo Marchiori and Vito Latora, *Harmony in the small-world*, Physica A: Statistical Mechanics and its Applications **285** (2000), no. 3-4, 539–546.

[MW21] Sakib Mostafa and Fang-Xiang Wu, *Diagnosis of autism spectrum disorder with convolutional autoencoder and structural MRI images*, Neural engineering techniques for autism spectrum disorder, Elsevier, 2021, pp. 23–38.

[NKJM11] Lakshmanan Nataraj, Sreejith Karthikeyan, Gregoire Jacob, and Bangalore S Manjunath, *Malware images: visualization and automatic classification*, Proceedings of the 8th international symposium on visualization for cyber security, 2011, pp. 1–7.

[PANB22] Dorottya Papp, Gergely Acs, Roland Nagy, and Levente Buttyán, *SIMBIoTA-ML: Light-weight, Machine Learning-based Malware Detection for Embedded IoT Devices*, IoTBDS, 2022, pp. 55–66.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al., *Pytorch: An imperative style, high-performance deep learning library*, Advances in neural information processing systems **32** (2019).

[Pri24] Simon J.D. Prince, *Understanding Deep Learning*, MIT press, 2024.

[SH12] Michael Sikorski and Andrew Honig, *Practical malware analysis: the hands-on guide to dissecting malicious software*, no starch press, 2012.

[SNB23] József Sándor, Roland Nagy, and Levente Buttyán, *Increasing the Robustness of a Machine Learning-based IoT Malware Detection Method with Adversarial Training*, Proceedings of the 2023 ACM Workshop on Wireless Security and Machine Learning, 2023, pp. 3–8.

[Spe04] Charles Spearman, *The proof and measurement of association between two things*, The American Journal of Psychology **15** (1904), no. 1, 72–101.

[SRKC16] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero, *Avclass: A tool for massive malware labeling*, Research in Attacks, Intrusions, and Defenses: 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings 19, Springer, 2016, pp. 230–253.

[Tah18] Rabia Tahir, *A study on malware and malware detection techniques*, International Journal of Education and Management Engineering **8** (2018), no. 2, 20.

[Tea17] Radare2 Team, *Radare2 Book*, GitHub, 2017.

[TPB21] Csongor Tamás, Dorottya Papp, and Levente Buttyán, *SIMBIoTA: Similarity-based Malware Detection on IoT Devices*, IoTBDS, 2021, pp. 58–69.

[WBC+23] Chia-Yi Wu, Tao Ban, Shin-Ming Cheng, Takeshi Takahashi, and Daisuke Inoue, *IoT malware classification based on reinterpreted function-call graphs*, Computers & Security **125** (2023), 103060.

[WDZ+21] Yueming Wu, Shihan Dou, Deqing Zou, Wei Yang, Weizhong Qiang, and Hai Jin, *Obfuscation-resilient android malware analysis based on contrastive learning*, arXiv e-prints (2021), arXiv–2107.

[WDZ+22] _____, *Contrastive Learning for Robust Android Malware Familial Classification*, IEEE Transactions on Dependable and Secure Computing, 2022, pp. 1–14.

[WF94] Stanley Wasserman and Katherine Faust, *Social network analysis: methods and applications*, Cambridge University Press (1994), 200–201.

[xor15] xorpd, *FCatalog*, `https://www.xorpd.net/pages/fcatalog.html`, 2015, last accessed: 2024-02-16.

[YY10] Ilsun You and Kangbin Yim, *Malware obfuscation techniques: A brief survey*, 2010 International conference on broadband, wireless computing, communication and applications, IEEE, 2010, pp. 297–300.

[ZLLS23] Aston Zhang, Zachary C Lipton, Mu Li, and Alexander J Smola, *Dive into deep learning*, Cambridge University Press, 2023.

[ZS18] Zhilu Zhang and Mert Sabuncu, *Generalized Cross Entropy Loss for Training Deep Neural Networks with Noisy Labels*, Advances in neural information processing systems **31** (2018), 8792—-8802.

[ZW20] Xiao Zhang and Dongrui Wu, *Empirical studies on the properties of linear regions in deep neural networks*, International Conference on Learning Representations, 2020.