Detecting Similar Malicious Payloads over Encrypted Data

Sophia Theresa Pietsch
University of Waterloo

Ehsan Amjadian
RBC & University of Waterloo

Florian Kerschbaum
University of Waterloo

Abstract

TODO: Abstract

1 Introduction

Anomaly detection systems provide very useful information in narrowing down threats that are undetected by pattern matching systems. However, they are intrinsically restricted to the information available to a single party, since the information analyzed is often confidential or personally identifiable. Once the information has been analyzed and properly sanitized threat information is commonly shared that can be incorporated into a pattern matching system. However, this process can take several months to complete and threats can spread much more quickly.

In this paper we present an architecture that aims to provide an early signal that is spread across multiple parties. In order to do so, we encrypt the information before sharing and perform similarity comparison over encrypted data. Our architecture consists of several steps. First, each party uses a common autoencoder trained on public samples in order to encode the payload into a fixed-length format that preserves similarity. This autoencoder forms the core of our architecture and is presented in this paper. Second, each party encrypts their payload using inner-product encryption [15] and sends it to an aggregator. Third, the trusted aggregator performs similarity comparison and flags anomalous events to the submitters. In this paper we focus on PDFs as payload as the first example, since they have been used as malware in the past and are usually too sensitive to share in plain.

Autoencoders have been previously used to perform anomaly detection [6, 21, 25]. The idea is to train the autoencoder on benign (normal) samples and use the reconstruction loss as an indicator of anomaly. However, implementing this privacy-preservingly would require to run the decoder of the autoencoder over encrypted data revealing the reconstruction loss. This is computationally inefficient. Instead, we use the autoencoder to produce an encoding which can use for distance-based clustering algorithms. This requires a different autoencoder architecture. First, our autoencoder should represent the entire space (benign and malicious samples) with sufficient distance between each sample. We introduce a specific loss term to achieve this objective during training. Second, our autoencoder should be robust to easy-to-perform modifications by the adversary. Malicious PDF clasifiers have been easily evaded in the past [10, 19]. We use adversarial training [11] and evaluate it on modifications not considered during training. Third, our autoencoder needs to produce an encoding for which we can compute the distance over encrypted data. We use inner-product encryption by Kim et al. [15] because the aggregrator can operate non-interactively (and only one aggregator is needed). However, we require the encoding to be a vector of integers from a small set. We use a differentiable version of rounding during training to improve accuracy.

In summary, we contribute a new architecture for privacy-preserving anomaly detection that features a new autoencoder that

- covers the entire input space of benign and malicious samples (PDFs),
- is robust to easy-to-perform modifications by the adversary, and
- produces an output can be used to compute distances over data encrypted by inner-product encryption.

The remainder of the paper is structured as follows ...

2 Background

2.1 Autoencoder

An autoencoder is a neural network where the outputs are trained to match the input as closely as possible. In autoencoders, outputs may not exactly match the inputs even after training the network. This is due to the noise inevitably injected by the architecture since a single network is tasked to learn patterns scattered across numerous data points. The amount of noise can increase if architectural components such as hidden layers with low dimensionality, called bottlenecks, or regularization are used. Autoencoders are employed for various tasks, including feature extraction, dimensionality reduction, generating synthetic/noisy data or denoising.

Autoencoders consist of two parts: an encoder function $H = E(X)$ that returns some hidden representation H of the input X and a decoder function $\overline{X} = D(H)$ that returns some approximation $\overline{X}$ of X. Both E and D can be deep neural networks. Given some distance measure d, the model is trained through backpropagation using the loss function $L(X,\overline{X}) = d(X,\overline{X})$. Depending on the task, either the hidden representation H or the reconstructed input $\overline{X}$ is the desired result. In our architecture, we use an autoencoder to reduce the dimensionality of the feature space. The hidden representations $\overline{X}$ of the trained autoencoder are encrypted and compared in the anomaly detection system. The space of all hidden representations is called the latent space.

2.2 Inner Product Encryption

The following description is based on the paper "Function-Hiding Inner Product Encryption is Practical" [15]. A functional encryption scheme is a scheme where functions f have associated secret keys. Given a ciphertext of a message x and a secret key for a function f, decryption gives the value $f(x)$ without leaking any other information about x. In inner product encryption, all messages $\vec{y} \in \mathbb{Z}_q^n$ are vectors of length n with all elements being integers in the range $[0, q-1]$. The functions are of the form $f_{\vec{x}}(\vec{y}) = \langle \vec{x}, \vec{y} \rangle$ for some message $\vec{x} \in \mathbb{Z}_q^n$. Hence, on decryption with the secret key for a function $f_{\vec{x}}$ and the ciphertext for a message $\vec{y}$, the inner product of $\vec{x}$ and $\vec{y}$ is returned. Note that decryption takes longer for larger values of q. An inner product encryption scheme is called function-hiding if the value of $\vec{x}$ cannot be determined from the secret key for the function $f_{\vec{x}}$.

The encryption scheme has a master secret key, which can be used to generate secret keys for functions $f_{\vec{x}}$ given a vector x and to generate ciphertexts given a message y. Note that the master secret key is not required for decryption.

2.3 Locality-sensitive hashing

Locality sensitive hashing (LSH) is an algorithm for finding similar datapoints according to some distance metric. The algorithm is based on the LSH family of hash functions. A hash function in the family has a higher collision probability for elements that are close according to the distance metric. The LSH algorithm uses n such hash functions on each element and inserts each element into the resulting buckets of n hash tables. The algorithm returns all elements that are in the same bucket as some element p in any of the n hash tables. The returned elements have a high probability of being close to element p according to the distance measure [24].

The particular hash functions used are chosen based on the distance metric and based on the desired maximum distance between close elements [24]. The Hamming distance is the number of positions at which the bits of two vectors are different. For Hamming distances over vectors in $\{0,1\}^d$, hash functions from the family $H = \{h_i : h_i(b_1 b_2 \dots b_d) = b_i, i = 1, \dots, d\}$ can be used. These functions can be combined to form hash functions that sample some combinations of bits from the d-dimensional vectors [13].

In our architecture, we use LSH to reduce the number of comparisons the aggregator must compute over encrypted data.

2.4 Portable Document Format

2.4.1 Structure of PDF Files

The information in this section is based on the PDF Reference [5].

A Portable Document Format (PDF) file contains of four sections: The header, the body, the reference table and the trailer. The header determines which version of the PDF standard is used, while the body contains objects that define the contents of the PDF. The reference table gives the positions of objects in the body and the trailer contains the position of this table in the file.

There are eight different types of objects. The primitive object types are Boolean values, Numbers, Strings, Names and Null. Here, Numbers can be either an Integer object or a Real object. Strings are a sequence of bytes enclosed by delimiting parentheses. Similarly, a Name object is a sequence of characters. Unlike Strings, Names are treated as atomic values and cannot contain any white-space characters. The remaining object types all contain references to other PDF objects. An Array is a sequence of objects enclosed in square brackets. A Dictionary maps keys to values, where keys are Name objects and values can be any kind of object. Finally, a Stream object contains a dictionary describing the properties of the stream as well a a sequence of bytes of un-

limited length, delimited by stream and endstream keywords.

The root object in the body of a PDF file is the Catalog dictionary. This dictionary contains references to objects that define the contents of the document and how the document is displayed.

2.4.2 Malicious PDF Files

PDF Readers can execute JavaScript specified in the PDF content, triggered by certain events such as opening the document. The JavaScript can perform specific actions, such as executing commands on the operating system, through an API defined in the Acrobat Forms JavaScript Object Specification [4]. In the document, the JavaScript code is contained in String or Stream objects. These objects are referenced inside action dictionaries that determine when the code is executed.

While many actions require user confirmation, vulnerabilities in the PDF reader can be exploited to bypass confirmation or change the confirmation text. Additionally, PDF documents can contain attached files that are automatically opened by the PDF reader. These embedded files can be used to hide a malicious PDF inside a benign one. Finally, many PDF readers do not require PDFs to follow the standard correctly. To detect malware, a sufficiently flexible parser must be used.

3 Architecture

3.1 Overview

While the majority of previous work attempts to classify PDFs as benign or malicious [26, 16, 20], we aim to create an architecture to detect anomalous PDF files. Many existing classifiers can be evaded using an adversarial attack that shifts the maliciousness score of a PDF towards zero [19]. By utilizing a larger feature space on which to cluster PDFs, we hope to be more robust to adversarial attacks.

Our second main objective is constructing an architecture that can be shared between multiple parties. This necessitates the encryption of all data sent to the shared part of our architecture. We use an autoencoder and locality sensitve hashing to the reduce the costs of comparisons over encrypted data points.

Figure 1 shows the steps required to train our autoencoder. Based on a dataset of PDFs, we use feature extraction, data augmentation and a learning step to obtain our trained autoencoder.

During feature extraction, we use existing tools [26, 23] to extract structural elements and JavaScript tokens from the PDF files. Then, these characteristics are transformed into real-valued feature vectors. In the data augmentation phase, we split the dataset into training, test and validation sets. We add modified versions of feature vectors corresponding to small changes to PDF files to the datasets. The modified versions let us train the autoencoder to be robust to small changes, which we hope makes the architecture robust against adversarial attacks.

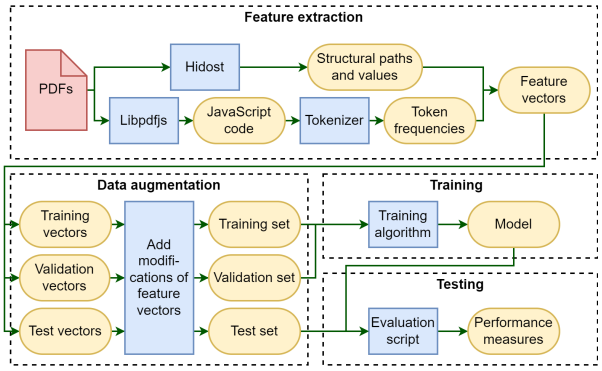

Figure 1: The architecture for training our autoencoder. The feature extraction step converts PDFs to feature vectors. Additional feature vectors are created using data augmentation before the autoencoder is trained. The model is evaluated on a test set.

Next, our autoencoder is trained using two loss functions. The first loss function trains the autoencoder to reconstruct the input feature vectors, while the second loss function maximizes the distance between the hidden representations of unmodified feature vectors. The autoencoder is evaluated using the test dataset.

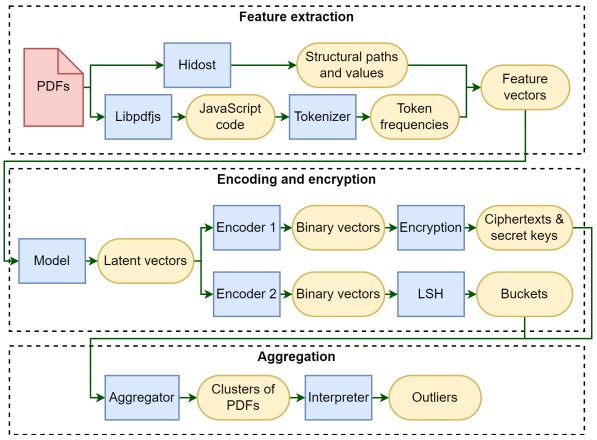

Figure 2: The architecture used for inference. After extracting features from the PDFs, we obtain latent representations from our autoencoder. In the encoding and encryption component we use both inner product encryption and locality-sensitive hashing on the latent vectors. Finally, the aggregator computes clusters that can be used to detect outliers.

The final architecture of our anomaly detection system is shown in Figure 2. After feature extraction, the feature vectors are encoded using the trained autoencoder. The resulting vectors are encoding, encypted and sent to the aggregator for clustering.

The autoencoder is designed to output a fixed-length vector with entries lying in a specified range. This format allows the encoding of the hidden representations into binary vectors so that the inner product of two such vectors returns the L1 distances of the hidden representations. These binary vectors are encrypted using inner product encyption and hashed using locality-sensitive hashing. The aggregator receives all the ciphertexts, secret keys and can use decryption to determine the inner product of any pair of binary vectors. This allows the aggregator to use the L1 distances between hidden representations that fall into the same buckets to compute clusters of similar PDFs.

3.2 Feature extraction

For the extraction of structural features, we use the hidost toolset [26]. The toolset uses the Poppler PDF parsing library [3] to extract all structural paths from the PDF. A structural path is a sequence of PDF objects starting with the Catalog dictionary, where each object in the sequence references the next object in the sequence. To prevent cycles, only the shortest structural path ending with each object is considered. A structural path is identified by the names of the objects in the sequence. Some semantically equivalent paths are combined according to the structural path consolidation rules given in "Hidost: a static machine-learning-based detector of malicious files". For instance, the rules replace user-defined object names with "Name" and remove repetitive subpaths [26].

Hidost assigns values to all remaining structural paths based on the contents of the last PDF object in the path. For Boolean objects, the value is 0 or 1. For Numbers, the value of the number is used. For Strings and Streams, the path is assigned the value 1. If one path results in multiple values, for instance if the last object in the path is an Array or there are multiple instances of a path in one file, the values are combined by finding their median [26]. Instead of using all paths found in the dataset, we select only paths that occur in at least one percent of the PDFs. We then use the selected paths and their assigned values as features.

To extract the JavaScript code from PDFs, we use the libpdfjs library [23]. This library starts at the Catalog and scans the hierarchy of PDF objects for action dictionaries. For each action dictionary found, libpdfjs determines whether the action is defined through JavaScript code. Should this be the case, the code is extracted from String or Stream objects. To tokenize the JavaScript code, we use the PyNarcissus JavaScript parser [14]. The frequencies of all tokens found in the training dataset are used as features.

The structural and JavaScript features are combined to form the final feature vectors.

3.3 Data augmentation

The data augmentation component creates additional data points that we use to train our autoencoder to be robust to certain changes in PDFs. We aim to reduce the effectiveness of adversarial attacks by directly including robustness to small changes in PDFs in the training process.

The difference between an initial malicious file and the file used in an adversarial attack can often be broken down into several small modifications or additions to the original file. Mostly, the original structure and content remains in the file, with additional added content masking its presence.

Hence, we aim to mitigate adversarial attacks by ensuring that small modifications and additions to PDF files only result in small changes to the hidden representations returned by our autoencoder. Then, our system would identify adversarial attacks as anomalies as the changes do not bring them much closer to benign PDF files.

To give us sufficient training data, we apply 25 small modifications to all PDF feature vectors. Each modification corresponds to some change to an arbitrary PDF file, such as adding an additional empty page or an image to the file. We determine the correct modifications to the feature vectors by analyzing how the changes to the PDF files impact the resulting feature vectors. Table 1 shows how many features each change impacts.

For our model to be robust to all adversarial attacks, the model needs to be robust to arbitrary small modifications. Since we only consider a small subset of possible changes, we evaluate our autoencoder on a subset of the modifications that it was not trained on. To do so, the 25 modifications are randomly split into three sets: 19 are assigned to the training set and 3 are assigned to each of the validation and test sets. For each data point in the training set, we include the data point as well as all modifications of that point that are assigned to the training set in the augmented training set. We similarly construct the augmented validation and test sets. Each data point is labeled as modified or unmodified.

3.4 Autoencoder architecture

For encryption and decryption to be sufficiently fast when clustering PDFs, the dimensionality of the vectors being clustered needs to be small. Furthermore, we want

Modification	# features changed
Add a page	1
Increase page size	1
Decrease page size	1
Increase image size	2
Decrease image size	2
Add an image	5
Display the document title	1
Add an outline	4
Add page labels	3
Add an annotation	4
Add a form	10
Specify the page layout	1
Open in fullscreen	3
Hide the toolbar	3
Create an alert	5
Open another file	4
Send an email	5
Change the colour of a form	6
Export a form to a file	6
Retrieve data from a URL	5
Align a field	7
Calculated field 1	9
Calculated field 2	11
Calculated field 3	12
Define options of a combobox	7

Table 1: The number of features that are changed for each modification

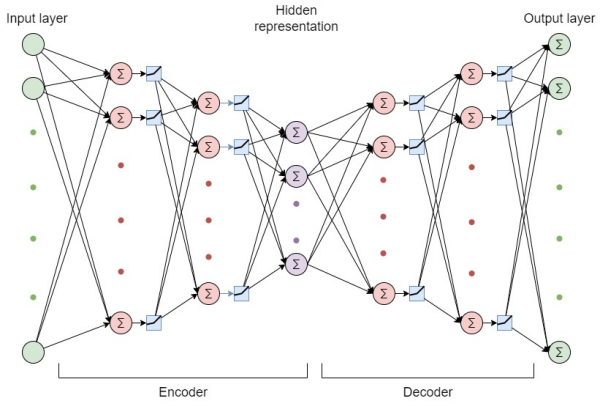

Figure 3: Autoencoder architecture

to train the data to be robust to the changes introduced in the data augmentation step. Our autoencoder is designed and trained to achieve both these tasks.

Figure 3 shows the structure of our autoencoder. The last layer of the encoder and decoder both use a linear activation function, all remaining layers use a leaky ReLu activation function. The hidden representation has 128 neurons.

The model is trained to reconstruct the inputs using a smooth L1 loss function. We add a custom secondary loss function to penalize short distances between unmodified feature vectors in the training set. By doing so, we hope that each unmodified feature vector and its modifications will result in a separate cluster in the latent space. We evaluate the impact of the secondary loss function using an ablation study. In our model, the loss function is computed for each batch of training data. Let S be the set of the hidden representations of unmodified PDFs in the current batch. The secondary loss function is computed using the following formula:

$$L(S) = \sum_{h_i \neq h_j \in S} \frac{1}{||h_i - h_j||_1}$$

Here, the L1 distance is used to measure the distance between two hidden representations. Notice that only the subset S of unmodified PDFs in each batch impacts the value of the loss function. Since the training set is augmented using 19 modifications in the data augmentation step, only 5% of each batch impacts the value of the loss function. This amplifies the amount of variance across batches. To mitigate the amount of variance, we use a considerably large batch size of 16,284 (2^{14}). The overall loss function used during training is a weighted sum of these two loss functions, with the weights optimized using the validation set.

To use inner product encryption in the next component of our architecture, the hidden representation needs to be a vector of integers in $[0, \ell - 1]$ for some integer ℓ. This rounding cannot be done exactly during training, as the derivative of the rounding function would be zero almost everywhere. A derivative of zero would prevent the model from using backpropagation to learn the model weights. Instead, we modify the autoencoder architecture to approximately round the values while still keeping nonzero derivatives (see Figure 4).

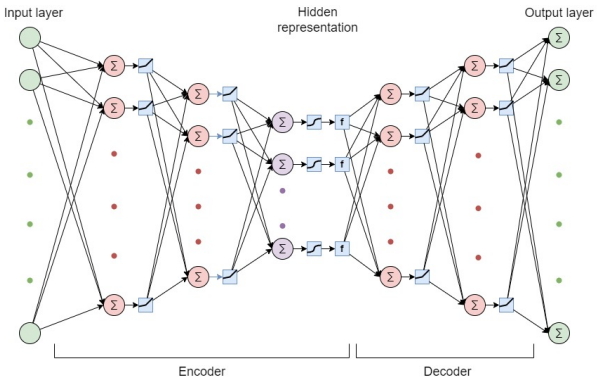

Figure 4: Autoencoder architecture after adding quantization

We replace the linear activation function in the last layer of the encoder with two activation functions. First, HardTanh function is used to restrict the values of the hidden layer to $[0, \ell-1]$:

$$\text{HardTanh}(x) = \begin{cases} \ell-1 & \text{if } x > \ell-1 \\ 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

Then, we use the following approximation of the rounding function:

$$f(x) = x - \frac{\sin(2\pi x)}{2\pi}$$

This approximation has large enough gradients to allow for backpropogation.

After training has completed, exact rounding is performed on the hidden representations of the feature vectors in the test set. Since the model is trained to return approximately rounded hidden representations, only a small amount of precision is lost when rounding exactly.

3.5 Encoding and encryption

We apply inner product encryption to vectors such that the inner product of the vectors is equal to the L1 distance of the rounded hidden representations corresponding to the vectors. To achieve this, each rounded hidden representation h is encoded as two different binary vectors. Let h_i be the ith dimension of the hidden representation h.

In the first binary representation, h_i is transformed into a binary vector a_i of length 2ℓ as follows:

$$a_i = (\underbrace{1,\ldots,1}_{\ell-h_i},\underbrace{0,\ldots,0}_{h_i},\underbrace{0,\ldots,0}_{\ell-h_i},\underbrace{1,\ldots,1}_{h_i})$$

In the second binary representation, the dimension h_i is transformed into a binary vector b_i of length 2ℓ as follows:

$$b_i = (\underbrace{0,\ldots,0}_{\ell-h_i},\underbrace{1,\ldots,1}_{h_i},\underbrace{1,\ldots,1}_{\ell-h_i},\underbrace{0,\ldots,0}_{h_i})$$

In both binary encodings, the encoding for the whole hidden representation h is obtained by concatenating the binary vectors for each dimension h_i:

$$a = (a_1, a_2, \ldots, a_n) \quad \text{and} \quad b = (b_1, b_2, \ldots, b_n)$$

Lemma 1: If a is the first binary encoding of a hidden representation h_1 and b is the second binary encoding of a hidden representations h_2, then

$$\langle a, b \rangle = ||h_1 - h_2||_1$$

Proof: If h_1 and h_2 have length n, then by the construction above a and b have length $2\ell n$. Hence, $\langle a,b\rangle = \sum_{i=1}^{2\ell n} a_i \cdot b_i = \sum_{j=0}^{n-1}\sum_{i=1}^{2\ell} a_{2\ell j+i} \cdot b_{2\ell j+i}$. For any $0 \le j < n$, $x = (a_{2\ell j+1}, a_{2\ell j+2} \ldots, a_{2\ell j+2\ell})$ is the first encoding of $h_{1_{j+1}}$ and similarly $y = (b_{2\ell j+1}, b_{2\ell j+2} \ldots, b_{2\ell j+2\ell})$ is the second encoding of $h_{2_{j+1}}$.

Now, we consider two cases. If $h_{1_{j+1}} \ge h_{2_{j+1}}$, then by definition $x_i \cdot y_i = 0$ for $1 \le i \le \ell$. For $\ell < i \le 2\ell$,

$$x_i \cdot y_i = \begin{cases} 1 & \text{if } \ell - h_{1_{j+1}} < i \le \ell - h_{2_{j+1}} \\ 0 & \text{otherwise} \end{cases}$$

It follows that $\sum_{i=1}^{2\ell} x_i \cdot y_i = h_{1_{j+1}} - h_{2_{j+1}}$. Similarly, when $h_{1_{j+1}} < h_{2_{j+1}}$ we get $\sum_{i=1}^{2\ell} x_i \cdot y_i = h_{2_{j+1}} - h_{1_{j+1}}$.

Thus, for $0 \le j < n$, $\sum_{i=1}^{2\ell} a_{2\ell j+i} \cdot b_{2\ell j+i} = |h_{1_{j+1}} - h_{2_{j+1}}|$

Thus, $\langle a,b\rangle = \sum_{j=0}^{n-1} |h_{1_{j+1}} - h_{2_{j+1}}| = ||h_1 - h_2||_1$. □

Lemma 1 shows that we can use inner product encryption to obtain the L1 distance between any two hidden representations. To do so, we compute the secret keys using the first binary encoding and the ciphertexts using the second binary encoding for all hidden representations h. To obtain the distance between h_1 and h_2, we decrypt using the secret key for the first encoding of h_1 and the ciphertext for the second encoding of h_2.

Using only inner product encryption, the aggregator will need to compute the pairwise distances between all hidden representations to determine a clustering. To reduce the number of comparisons and hence the decryption time, we use locality-sensitive hashing.

To use LSH on Hamming distances, we convert each hidden representation h as another binary vector. Here, we map h_i, the ith dimension of the hidden representation of h to c_i:

$$c_i = (\underbrace{1,\ldots,1}_{h_i},\underbrace{0,\ldots,0}_{\ell-h_i})$$

The encoding of of the whole hidden representation h is obtained by concatenating the binary vectors for each dimension h_i:

$$c = (c_1, c_2, \ldots, c_n)$$

Now, the L1 distance between two hidden representations h_1 and h_2 is given by the Hamming distance between their encodings $c^{(1)}$ and $c^{(2)}$. Thus, we can use LSH on the vectors c to obtain buckets for each hidden representation (see section 2.3). When computing clusters, we only need to compute distances for hidden representations that are both in some bucket.

Note that the Hamming distance and the L1 distance are the same in their respective encodings. Hence, the

aggregator learns no new information for the LSH except from LSH errors. The aggregator could have computed the Hamming distance for all close elements and performed the binning (for correctly hashed elements) from the encrypted vectors. The clients can keep the hash functions secret from the aggregator which further complicates exploiting the leakage from LSH errors. The exact L1 distance allows better clustering than just the LSH.

TODO: Distance thresholding and key management

3.6 Aggregation

The aggregator receives the secret keys and ciphertexts corresponding to hidden representations as well as the buckets from locality-sensitive hashing for each ciphertext. We use a clustering algorithm that can handle an arbitrary number of clusters and unknown shapes of clusters, since when deploying our architecture in practice we will not know the distribution of PDFs. To increase the chances of detecting outliers, we want the algorithm introduce new clusters instead of having points that are far from all other points in a cluster.

Hence, we choose the following clustering criteria for some threshold:

- Within each cluster, no element is further than the threshold away from all other elements.
- Between clusters, there are no two elements that are closer than the threshold to each other.

Note that this clustering algorithm fulfils the desired characteristics described above.

The distance between elements used in the algorithm is the L1 distance. As long as the distance threshold used in the locality-sensitive hashing is greater than or equal to the threshold used in the clustering algorithm, we only need to compute the exact distances for elements which hash to the same buckets. For such elements, we use the inner-product encryption to retrieve the L1 distance, as described in the previous section.

Since the aggregator only receives encrypted information, multiple parties can send their hidden representations to the same aggregator. Consequently, the aggregator has more samples to build clusters from, resulting in a more robust set of clusters. In addition to warning single parties about anomalies, the aggregator can quickly detect large-scale attacks. Should similar PDFs that fall outside the usual clusters be sent to multiple parties, the aggregator can send an early warning to all parties about the potential attack.

4 Evaluation

4.1 Data sources

The first data source used in this study is the Common Crawl corpus [1]. The Common Crawl corpus contains raw web page data as well as metadata obtained by web crawling. We scanned the July/August 2021 crawl archive for files with a MIME (Multipurpose Internet Mail Extensions) type of ”application/pdf” and downloaded them, resulting in a dataset of 74000 benign PDF files. We use exclusively this dataset for most of our experiments.

The second data source we used is a Contagio dataset designed for signature and security product testing. The data source includes various types of benign and malicious files, including 9100 benign and 11100 malicious PDFs. Some of the malicious PDFs are sorted by the specific vulnerability that is exploited [2]. We use the malicious files from this dataset to evaluate the effectiveness of our model in isolating malicious PDF files.

4.2 Experiments

To evaluate our model, we assess the clusters produced by the aggregator. To judge the different components of our autoencoder architecture, we evaluate different versions of the architecture seperately.

- First, we use an ablation study to determine the impact of the secondary loss function. For this experiment, we use the architecture from Figure 3. We anticipate that adding the secondary loss function improves the clusters by increasing the distance between clusters.
- The next experiments assesses the impact of quantizing the hidden representations. We compare the two architectures from Figure 3 and Figure 4. When using quantization, we test two different values of ℓ: $\ell = 10$ and $\ell = 25$. We anticipate that the quantization will decrease the quality of the clustering since we limit the latent space.
- TODO: Add experiment with malicious data

For the first three experiments, we use only the 74000 PDFs retrieved from the Common Crawl corpus. We randomly select 2500 PDFs for each of the test and validation sets, and use the remaining 69000 PDFs for training. In the data augmentation phase, we assign 3 modifications to the test set, 3 modifications to the validation set and use the remaining 19 modifications for training. Since the assignments are random, we run all experiments for 5 different assignments of the modifications

to obtain representative results. We evaluate the aggregation using various thresholds for the clustering criteria to vary number of clusters.

4.3 Metrics

We use two metrics to evaluate the clusters returned by the aggregator. First, we compute precision and recall. We introduce one label for each unmodified PDF in the dataset as well as one NULL label. We say that a PDF corresponds to the label if the PDF is the unmodified PDF or one of its modifications. These labels are assigned to clusters as follows:

- The cluster that contains the most PDFs corresponding to each label is assigned that label. If multiple clusters contain the same number of PDFs corresponding to a label, one cluster is assigned the label at random. If one cluster would be assigned several labels, one of the labels is chosen at random.
- If a cluster is not given a label through the rule above, then it is labeled NULL.

A PDF is counted as a true positive if the PDF corresponds to the label of the cluster it is in. If the cluster has the label of a different PDF, then the PDF is counted as a false positive. Finally, if the cluster is labeled NULL, then the PDF is counted as a false negative. Precision and recall as well as the F-score can be computed using the following formulas:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

$$\text{F-Score} = \frac{2}{1/\text{Recall} + 1/\text{Precision}}$$

Our second metric is more strict, allowing us to differentiate further between sets of clusters that perform well according to the metric described above. We call the metric strict precision and strict recall. These are computed using the formulas above using the following definition of strict true positives, strict false positives and strict false negatives:

- A strict true positive is a cluster that only contains one unmodified PDF and all its modifications.
- A false positive is a cluster that contains only variations of one PDF, but does not contain all of its variations.
- A false negative is variations of one PDF that are contained in the same cluster as variations of some other PDF. For example, if a cluster contains 4 variations of PDF A, 2 variations of PDF B and 1 variation of PDF C, this would count as 3 false negatives.

Note that these definitions use a much stricter definition of true positive, creating a more conservative metric.

4.4 Ablation study

Figure 5 shows the results of the ablation study with respect to the first, standard metric. we obtain very good results both with and without the secondary loss function.

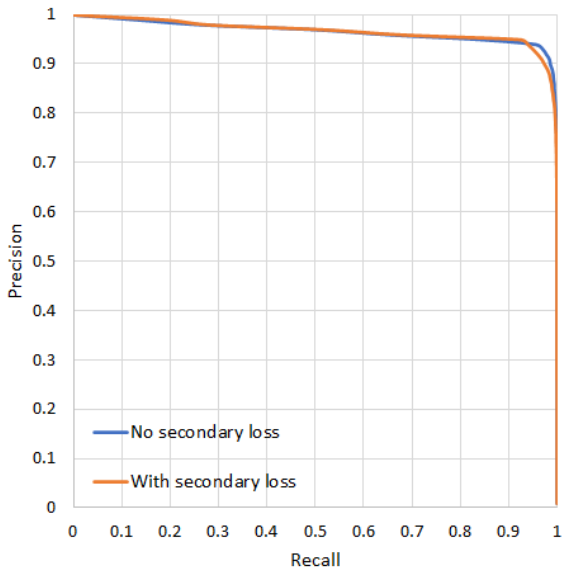

Figure 5: Precision-recall curve for our ablation study, using the standard metric. There is little visible difference when using the secondary loss function.

To be able to better differentiate between the results, we evaluate the results with respect to the strict metric, as shown in Figure 6. Note that for small thresholds, there are few true positives since there are many small clusters and most PDFs have their modifications split into multiple clusters. Hence, both precision and recall start low. For suitable thresholds, we achieve very good results both with and without the secondary loss function. Furthermore, results are consistently better when using the secondary loss function. Hence, in the following experiments we use the secondary loss function.

4.5 Quantization

4.6 Malicious PDFs

TODO: Malicious PDFs and results

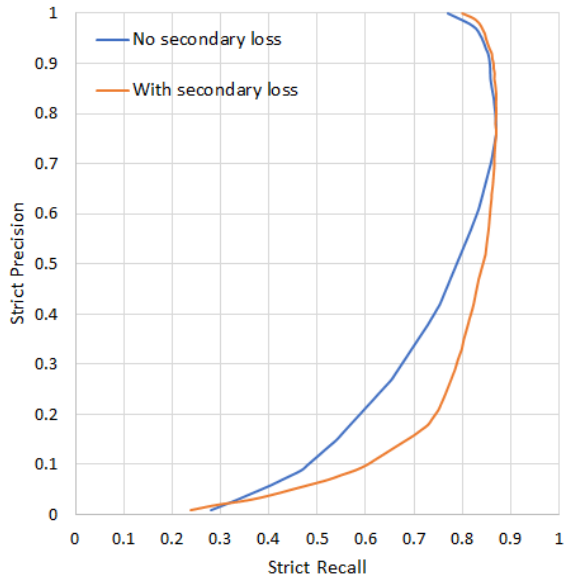

Figure 6: Precision-recall curve for our ablation study, using the strict metric. Here, the recall is better when using the secondary loss function.

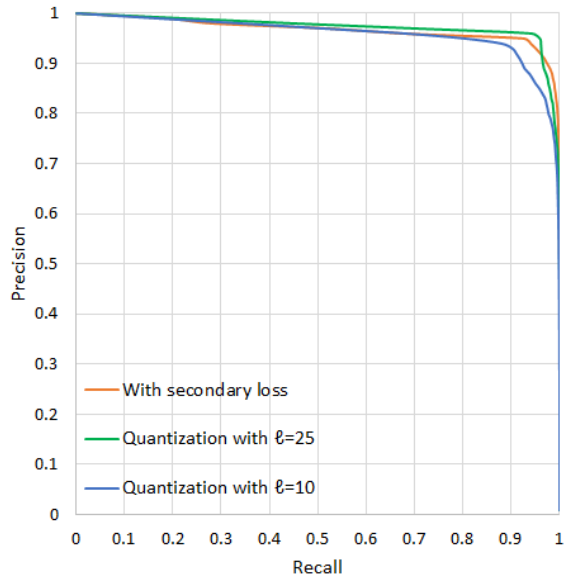

Figure 7: Precision-recall curve for our ablation study, using the strict metric. Here, the recall is better when using the secondary loss function.

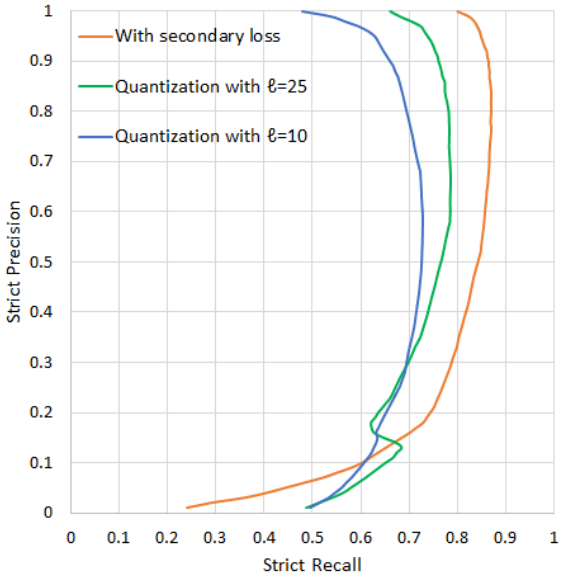

Figure 8: Precision-recall curve for our ablation study, using the strict metric. Here, the recall is better when using the secondary loss function.

4.7 Encryption

TODO: Microbenchmarks.

We measured the encryption (for sending encoded PDFs) and decryption (for comparing encoded and encrypted PDFs) time of inner-product encryption on our required vector sizes. Our encoding has 128 vector elements. We use integers in the range of either $[0,24]$ or $[0,99]$. Our encoding to compute the L1 distance doubles the vector length. Hence we consider vectors of lengths $128 \cdot 25 \cdot 2 = 6400$ and $128 \cdot 100 \cdot 2 = 25600$.

	6400	25600
Encryption	5s	20s
Decryption	19s	76s

Table 2: Caption

5 Related Work

Most previous works on PDFs focus on classifying PDFs as benign or malicious. Many models use structural and content-based information that is extracted from the PDFs in a preprocessing step.

Many models extract information from the PDF object hierarchy as features. Using structural paths and a random forest classifier, Hidost achieves a true positive rate of 99.73% [26]. Using a similar setup, PDF Malware Slayer [20] and PDFRate [22] achieve a true positive rate of 99.5% and 99.81% respectively. By hashing structural

information to an image and applying a Convolutional Neural Network, Liu et al. achieve a true positive rate of 96.9% [18]. The false positive rate of all these models is less than 0.1%.

Other models target JavaScript content to classify files. By using static analysis of JavaScript code with a Support Vector Machine, PJscan achieves a true positive rate of 71.94% and a false positive rate of 16.35% [16]. Combining static analysis with structural features gives better results. Dabral et al. use such a feature set with an ensemble of decision trees and achieve a true positive rate of 98.4% and false positive rate of 0.017% [9]. The performance of a JavaScript-based model can also be increased by using dynamic analysis, during which the JavaScript code is executed. Lux0R uses dynamic analysis and a random forest classifier to achieve a true positive rate of 99.27% and a false positive rate of 0.05% [8].

While these models achieve very good results on static datasets, most are vulnerable to evasion techniques. EvadePDF is an algorithm that modifies malicious PDF files to appear benign while keeping its malicious behaviour. The algorithm was able to evade detection by PDFRate 100% of the time [10]. Maiorca et al. inject malicious content into benign files to evade structural methods of classification. Their examples evade detection by PDFRate and PDF Malware Slayer, with both classifiers assigning a maliciousness score of under 0.2 out of 1 to all examples [19]. Approaches based on dynamic analysis are more resistant to evasion techniques.

Several different approaches have taken into account possible evasion techniques when constructing classification models. Liu et al. attempt to combat evasion techniques by building an ensemble of various different machine learning techniques, since an adversarial example is unlikely to avoid detection on all models simultaneously. The techniques used by EvadePDF and Maiorca et al. are not able to avoid detection in the resulting system [17]. Using a different approach, Chen et al. define distance metrics between two PDFs based on the tree structure of the PDFs objects. They use adversarially robust training to train a classifier that is robust to insertions and deletions in the object tree. While the robust model can still be evaded, 3.6 times as many changes are required to evade detection [7]. Finally, He et al. design a two-stage algorithm that first uses anomaly detection to identify the most unusual object of each type in a given PDF file and compute a distance measure for these objects. Next, a CNN combines the distances to obtain a classification for the PDF. The resulting model was able to correctly classify adversarial examples generated via mimicry attacks [12].

6 Conclusions

TODO: Conclusions

References

[1] Common crawl. `https://commoncrawl.org/`. Accessed: 2-03-2022.

[2] Contagio. `http://contagiodump.blogspot.com/`. Accessed: 2-03-2022.

[3] Poppler. `https://poppler.freedesktop.org/`. Accessed: 1-13-2022.

[4] ADOBE SYSTEMS INCORPORATED. Acrobat forms javascript object specification, 1999. `http://www.verypdf.com/document/acrobat-forms-javascript/index.htm`.

[5] ADOBE SYSTEMS INCORPORATED. Pdf reference third edition, 2001. `https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdfs/pdf_reference_archives/PDFReference.pdf`.

[6] AN, J., AND CHO, S. Variational autoencoder based anomaly detection using reconstruction probability. *Special Lecture on IE 2*, 1 (2015), 1–18.

[7] CHEN, Y., WANG, S., SHE, D., AND JANA, S. On training robust pdf malware classifiers, 2019.

[8] CORONA, I., MAIORCA, D., ARIU, D., AND GIACINTO, G. Lux0r: Detection of malicious pdf-embedded javascript code through discriminant analysis of api references. In *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop* (New York, NY, USA, 2014), AISec '14, Association for Computing Machinery, p. 47–57.

[9] DABRAL, S., AGARWAL, A., MAHAJAN, M., AND KUMAR, S. Malicious pdf files detection using structural and javascript based features.

[10] DEY, S., KUMAR, A., SAWARKAR, M., SINGH, P., AND NANDI, S. *EvadePDF: Towards Evading Machine Learning Based PDF Malware Classifiers*. 04 2019, pp. 140–150.

[11] GOODFELLOW, I. J., SHLENS, J., AND SZEGEDY, C. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[12] HE, K., ZHU, Y., HE, Y., LUI, L., LU, B., AND LIN, W. Detection of malicious pdf files using a two-stage machine learning algorithm, 2020.

[13] INDYK, P., AND MOTWANI, R. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1998), STOC '98, Association for Computing Machinery, p. 604–613.

[14] JTOLIO. Pynarcissus. `https://github.com/jtolio/pynarcissus`, 2015.

[15] KIM, S., LEWI, K., MANDAL, A., MONTGOMERY, H., ROY, A., AND WU, D. Function-hiding inner product encryption is practical.

[16] LASKOV, P., AND ŠRNDIĆ, N. Static detection of malicious javascript-bearing pdf documents. In *Proceedings of the 27th Annual Computer Security Applications Conference* (New York, NY, USA, 2011), ACSAC '11, Association for Computing Machinery, p. 373–382.

[17] LIU, C., LOU, C., YU, M., YIU, S., CHOW, K., LI, G., JIANG, J., AND HUANG, W. A novel adversarial example detection method for malicious pdfs using multiple mutated classifiers. *Forensic Science International: Digital Investigation 38* (2021), 301124.

[18] LIU, C.-Y., CHIU, M.-Y., HUANG, Q.-X., AND SUN, H.-M. *PDF Malware Detection Using Visualization and Machine Learning*. 07 2021, pp. 209–220.

[19] MAIORCA, D., CORONA, I., AND GIACINTO, G. Looking at the bag is not enough to find the bomb: An evasion of structural methods for malicious pdf files detection. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2013), ASIA CCS '13, Association for Computing Machinery, p. 119–130.

[20] MAIORCA, D., GIACINTO, G., AND CORONA, I. A pattern recognition system for malicious pdf files detection. vol. 7376, pp. 510–524.

[21] SAKURADA, M., AND YAIRI, T. Anomaly detection using autoencoders with nonlinear dimensionality reduction. In *Proceedings of the MLSDA 2014 2nd workshop on machine learning for sensory data analysis* (2014), pp. 4–11.

[22] SMUTZ, C., AND STAVROU, A. When a tree falls: Using diversity in ensemble classifiers to identify evasion in malware detectors. In *NDSS* (2016).

[23] SRNDIC, N., AND LASKOV, P. libpdfjs. `https://sourceforge.net/p/libpdfjs/home/Home/`, 2013.

[24] WANG, J., SHEN, H. T., SONG, J., AND JI, J. Hashing for similarity search: A survey. *ArXiv abs/1408.2927* (2014).

[25] ZHOU, C., AND PAFFENROTH, R. C. Anomaly detection with robust deep autoencoders. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining* (2017), pp. 665–674.

[26] ŹRNDIĆ, N., AND LASKOV, P. Hidost: A static machine-learning-based detector of malicious files. *EURASIP J. Inf. Secur. 2016*, 1 (dec 2016).