Dos and Don'ts of Machine Learning in Computer Security (Supplementary Materials)

1 Prevalence Analysis Details

In this section, we provide additional details on our prevalence analysis regarding the chosen papers and the author survey.

Distribution of selected papers. For the prevalence analysis, we have selected papers published in the last ten years at the leading four security conferences. Figure 1 shows a breakdown of these papers by year of publication. While the papers date back to 2011, the majority of work has been published between 2015 and 2019.

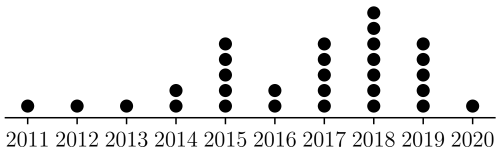

Figure 1: Distribution of papers per year for the 30 papers in our analysis.

2 Impact Analysis Details

This section provides additional information about the experiments in Section 4 of the paper.

2.1 Mobile Malware Detection

Here we describe the additional dataset used in our experiments and detail the experimental setup considered in §4.1 of the paper.

Analysis of the Drebin dataset. In addition to AndroZoo [2], we also analyze the meta information of the DREBIN [3] dataset that has been provided to us by the authors of the paper. Interestingly, we find that 76.2 % of the benign data has been collected from GooglePlay, while the fraction of malicious data is only 4.6 %. Although the origins for the majority of malicious samples is unknown (86.9 %), our findings strongly suggest the presence of a sampling bias in this dataset as well.

Experimental setup. While we have reimplemented the feature extraction of DREBIN [3], for OPSEQS [9] we use the publicly available program code as provided by McLaughlin et al. [9] to extract opcode *n*-grams. Using the extracted features, we represent each app as a binary vector and train a linear SVM [5] on the dataset. We use 75 % of the data for training and the remaining 25 % for testing. To select good hyperparameters for our classifiers, we perform a grid search on the training data for $C = \{10^{-2}, 10^{-1}, ..., 10^{2}\}$ and $n = \{2,\ 3,\ 4\}$ using 5-fold cross validation, where n refers to the length of the opcode *n*-grams. Finally, we assess the performance of the best model on the test data. We repeat the experiments ten times and average the results.

2.2 Vulnerability Discovery

We give additional information on the model and the evaluation methodology used for our experiments on vulnerability discovery in §4.2 of the paper.

Models and preprocessing. For VulDeePecker [8], we train a neural network consisting of a bidirectional LSTM layer with 300 units that is followed by a dropout layer with a probability of 0.5 and a dense layer of size 2 employing a softmax non-linearity. We use the Adam optimizer [7] with a batch size of 64 and train for 10 epochs (the network begins to overfit the training set after ~6 epochs). These hyperparameters for the architecture and training are adopted from the work of Li et al. [8] and not tuned explicitly.

The code snippets are preprocessed as described by Li et al. [8] and a word2vec [10] embedding of 200 dimensions is trained for 100 iterations to achieve vector representations of the generic code tokens. Word2vec models are solely determined based on the training data. Unknown tokens that occur at test time are replaced with a vector of zeros—the same value that is used to pad code snippets shorter than 50 tokens.

For the linear SVM, we use a regularization cost of $C = 1.0$ and token-level *n*-grams extracted from the generic tokens of the training data. The 3-grams obtained by this approach are used as input for the *AutoSklearn* framework [6]. Here we optimize the bounded area under ROC curve (FPR < 0.05) and limit the number of models in the ensemble to 50.

Performance evaluation. To compare the performance of VulDeePecker and the baseline models, we split the data into a randomly chosen training set (80 %), validation set (10 %), and test set (10 %) for 10 trials. All methods learn on the training set only and we use the model that performs best on the validation set. Finally, we compute ROC curves on the test data also containing unseen data instances and average the results over the 10 individual trials. The results are presented in Table 4 of §4.2. Note that picking an optimal threshold from these ROC curves is a form of data snooping (P3). In this case, however, we only use the ROC curves to compare the three classifiers on unseen data.

2.3 Authorship Attribution

Here we provide further intuition on the problem of artifacts in datasets for authorship attribution and describe the experimental setup used in §4.3 in more detail.

Artifact examples. Figure 2 exemplifies how attribution methods exploit features from copied code. The selected author copies both arrays in all files but never uses them. It turns out that the AST feature '`1`' is one of the most important features for classifying this author. However, these copied arrays are unrelated to the programming task and thus only loosely related to coding style in practice.

```
constexpr int dx[] = {-1, 0, 1, 0, 1, 1, -1, -1};
constexpr int dy[] = {0, -1, 0, 1, 1, -1, 1, -1};
```

Figure 2: Artifact example from the code GCJ dataset. Arrays are unused, but present in all files by the same author.

Experimental setup. For our evaluation of the attribution methods by Caliskan et al. [4] and Abuhamad et al. [1], we use a publicly available reimplementation [12] built on top of Clang. We also use a stratified and grouped 8-fold cross-validation where the dataset is divided into seven challenges for training and one challenge for testing, respectively. To select hyperparameters in each fold, we further perform a grid search on the training set using 3-fold stratified and grouped cross validation. We perform feature selection and a tf-idf transformation where we derive the parameters from the respective training set. Finally, we measure the accuracy of the best performing model on the test set. We report results for all eight folds in Figure 7 of §4.3, as the difficulty of attribution can vary across the GCJ challenges.

Reproducing the setup of Caliskan et al. [4], we use a random forest with layout, lexical and syntactical features. For Abuhamad et al. [1], we use the originally proposed features consisting of word *n*-grams, but apply a random forest only rather than a combination of recurrent neural network and random forest. We find that this leads to a comparable accuracy and has the benefit of a simpler analysis of each features' contribution to the classification.

Furthermore, we implement small linter tools in Clang that remove the following five groups of unused code in our experiments: functions, local and global declarations, typedefs, records, and headers.

2.4 Network Intrusion Detection

We provide details on the experimental setup as used for the case study on network intrusion detection described in §4.4 of the paper.

Experimental setup. For training the ensemble of autoencoders, we follow the procedure of KITSUNE [11]. The 115 features are derived from seven damped incremental statistics describing packet relationships of five time windows of up to a one minute interval. To determine a suitable number of autoencoders, we apply a hierarchical clustering to the first 5,000 examples from the feature set, resulting in a maximum of $m = 10$ inputs per autoencoder. Overall, this corresponds to 12 autoencoders in parallel. The outputs of these autoencoders are passed to another, final autoencoder which operates as the anomaly detector. The root mean squared error (RMSE) representing the autoencoders' reconstruction error is output for each packet individually. Consequently, we apply a threshold to the RMSE values, depending on how many false positives can be tolerated.

For both methods, the first 50,000 packets are used for training and the remaining 714,136 packets for testing. This corresponds to the first 30.5 and 80.4 minutes of the network packet capture, respectively.

References

[1] M. Abuhamad, T. AbuHmed, A. Mohaisen, and D. Nyang. Large-scale and language-oblivious code authorship identification. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2018.

[2] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proc. of the Int. Conference on Mining Software Repositories*, 2016.

[3] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck. Drebin: Efficient and explainable detection of Android malware in your pocket. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2014.

[4] A. Caliskan, R. Harang, A. Liu, A. Narayanan, C. R. Voss, F. Yamaguchi, and R. Greenstadt. De-anonymizing programmers via code stylometry. In *Proc. of USENIX Security Symposium*, 2015.

[5] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classi-

fication. *Journal of Machine Learning Research (JMLR)*, 9, 2008.

[6] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems (NIPS)*, 2015.

[7] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *Proc. of the International Conference on Learning Representations (ICLR) (Poster)*, 2015.

[8] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2018.

[9] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, and G. Joon Ahn. Deep android malware detection. In *Proc. of ACM Conference on Data and Applications Security and Privacy (CODASPY)*, 2017.

[10] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. In *Proc. of the International Conference on Learning Representations (ICLR) (Workshop Poster)*, 2013.

[11] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2018.

[12] E. Quiring, A. Maier, and K. Rieck. Misleading authorship attribution of source code using adversarial learning. In *Proc. of USENIX Security Symposium*, 2019.