Is It Overkill? Analyzing Feature-Space Concept Drift in Malware Detectors

Zhi Chen*, Zhenning Zhang*, Zeliang Kan†‡, Limin Yang*, Jacopo Cortellazzi†‡
Feargus Pendlebury‡, Fabio Pierazzi†, Lorenzo Cavallaro‡, Gang Wang*
*University of Illinois at Urbana-Champaign †King's College London ‡University College London

***Abstract*—Concept drift is a major challenge faced by machine learning-based malware detectors when deployed in practice. While existing works have investigated methods to detect concept drift, it is not yet well understood regarding the main causes behind the drift. In this paper, we design experiments to empirically analyze the impact of feature-space drift (new features introduced by new samples) and compare it with data-space drift (data distribution shift over existing features). Surprisingly, we find that data-space drift is the dominating contributor to the model degradation over time while feature-space drift has little to no impact. This is consistently observed over both Android and PE malware detectors, with different feature types and feature engineering methods, across different settings. We further validate this observation with recent online learning based malware detectors that incrementally update the feature space. Our result indicates the possibility of handling concept drift without frequent feature updating, and we further discuss the open questions for future research.**

1. Introduction

In recent years, machine learning (ML) models have been used to build malware detectors by researchers and practitioners [1, 2, 3, 4, 5]. Concept drift (i.e., the shift of ML decision boundaries) is a crucial challenge faced by these detectors when deployed in practice. Fundamentally, ML models expect the testing data to roughly match that of the training data. However, due to organic changes or adversarial adaptations, the testing samples' distribution can drift away, decreasing model performance over time [6].

Researchers have studied concept drift in malware detectors by proactively detecting drifting samples [7, 8, 9, 10] or aging models [11, 12]. However, less effort has been investigated to understand the contributors (causing factors) of concept drift. To fill this gap, in this paper, we aim to isolate the impact of two types of drift, namely, *feature-space* drift and *data-space* drift, and measure their influence on the model behavior changes. The goal is to provide a deeper understanding of the causes of concept drift and inform future designs to counter this effect.

Motivation Example. Suppose an ML model is trained for ransomware detection. Fig. 1 illustrates the difference between feature-space and data-space drifts. The top figure shows the data-space drift only. The model considers a fixed set of two features (f_1 is "*number-of-writes*" and f_2 is "*call-graph-complexity*"). This predefined feature set never changes. At t_1, the decision boundary is noted as the red dashed line. At t_n, there is a shift in both ransomware and benign behavior under these two features, which can lead to changes in the decision boundary and classification errors.

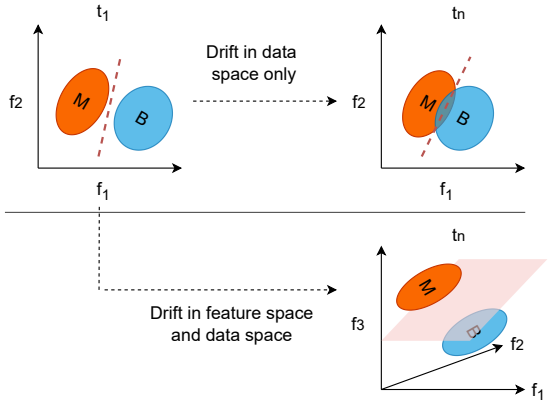


Figure 1: Motivation Example.

The bottom figure shows feature-space drift. Instead of fixing the feature set, the model actively adds new features that are never seen in the past. For example, recent ransomware starts to use a new cryptography library called "X". As such, a new feature f_3 ("*crypto-lib-X-is-present*") is introduced to the feature space at t_n. Under this new feature space (f_1, f_2, f_3), we can observe that there exists a hyperplane to cleanly separate the two distributions, thanks to the added feature dimension f_3.

The hypothesis is that newly arrived data may carry new features (previously unseen), which can be used to update the feature space to catch up with concept drift. This idea has been incorporated in online learning-based malware detectors [11, 12, 13] where new features are dynamically added to the feature space.

Our Work and Findings. In this paper, we designed experiments to empirically analyze the impact of feature-space drift in comparison with pure data-space drift in common malware detectors. We used datasets of both Android malware (7 years) and PE malware (2 years) to compare model performance *with* and *without* feature-space updating.

Surprisingly, we find that feature-space drift has minimal impact, compared with pure data-space drift (i.e., when features are completely fixed). For example, we construct a

2023 IEEE Security and Privacy Workshops (SPW) | 979-8-3503-1236-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/SPW59333.2023.00007

DOI 10.1109/SPW59333.2023.00007

feature set using data from the first quarter of 2015—without any changes to this feature set, the model performance is similar to (or slightly better than) models that actively incorporate newly appeared features in the data over 7 years of the testing period. This is consistently observed in both Android and PE detectors despite their significant differences in feature types and embedding methods. It also remains consistent under various model types, training data ratios, and feature set sizes. Despite the significant number of new features (millions) introduced by the new data over time, we show it is possible to stick to a fixed set of features to support model updating over many years.

We further measure the impact of feature-space updating on online learning-based malware detectors. Interestingly, we show that feature-space updating has little impact when the model is updated with ground-truth labels. However, feature-space updating starts to have an impact when the model is updated with noisy (less accurate) labels. Based on these observations, we discuss the implications of our results and future directions to study feature updating strategies to counter concept drift for malware detection.

2. Background and Related Work

Concept Drift in Machine Learning (ML). Concept drift is a general challenge for ML models. After a trained model is deployed, the testing data distribution may (gradually) shift away from that of the training data over time. Such concept drift can lead to model degradation [6, 14]. To detect concept drift, ML researchers have proposed various methods to statistically assess model behaviors and measure distribution changes [15, 16, 17, 18]. However, most of these methods require collecting extensive labels for the new data, which can be challenging for security applications.

Concept Drift in Malware Detection. Machine learning has been used for malware detection [1, 2, 3, 4, 5] and malware family classification [19, 20]. Concept drift also poses a challenge to these models, which requires the models to be periodically re-trained [21, 22]. Recent works propose to proactively detect drifting samples and use a smaller set of drifting samples for model updating [7, 8, 9, 10], which can reduce the overhead of data labeling.

Feature-Space vs. Data-Space Drift. While most existing works focus on concept drift detection [7, 8, 9, 10], less effort is investigated to *reason the causes of the drift*. We focus on *feature-space drift* and compare it with *data-space drift*. For data-space drift, suppose a malware classifier uses a fixed set of features F, concept drift happens when the data distribution $D(F)$ changes over this feature set. Feature-space drift additionally considers the changes in the underlying feature space ($F \rightarrow F'$) which then leads to new data distribution ($D(F')$). Note that feature-space drift always leads to data-space drift because data distribution $D(F')$ is dependent on feature space F'.

In the context of malware detection, feature-space drift describes the situation where new samples introduce new features (e.g., APIs, strings, libraries) that are never seen in the previous data. When such features appear, the model can either choose to ignore them or incorporate them. For example, CASANDRA [13] and DroidEvolver [11] adopted online learning algorithms to incrementally add features to the feature space as new samples arrive. Another recent system APIGraph [23] proposes to enhance this process by learning a robust feature space to reduce the frequency of feature updating. Our work is built on top of these existing works with a focus on (a) empirically comparing the impact of feature-space drift and data-space drift and (b) exploring the reasons behind the observed drift.

3. Method

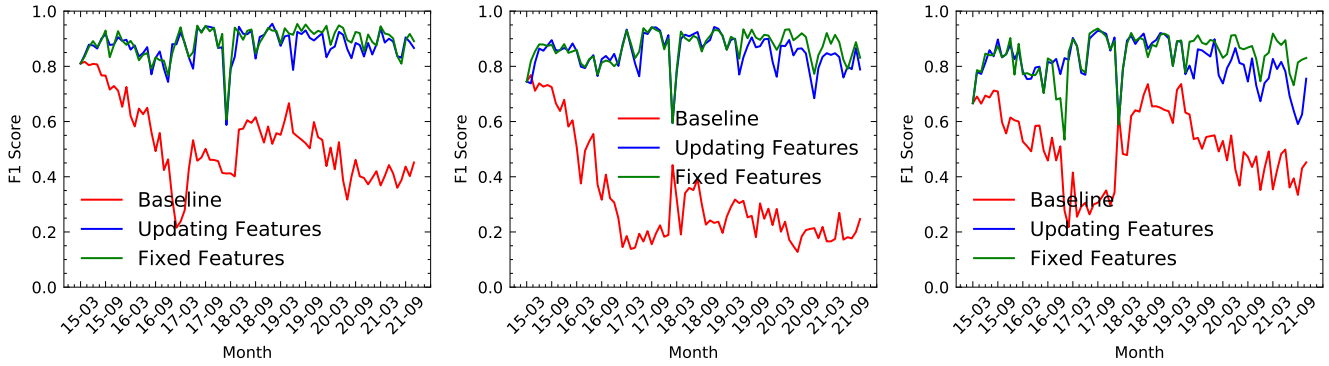
The goal of our analysis is to decouple the impact of feature space drift with that of data-space drift to understand the main contributor of concept drift. To do so, we take a malware dataset and divide it into n timestamped data blocks $[D_{t_1}, D_{t_2}, ..., D_{t_n}]$. Here, "data block" is just a general concept. In actual experiments, they can be generated with a non-overlapping data split or using a sliding time window.

1) **Measure data-space drift only**: we construct a feature space F_1 based on the first data block D_{t_1}. Then we fix this feature set F_1 for all the following data blocks. This feature set will be used for all the model training/re-training and testing for the following data blocks.
2) **Measure feature-space drift**: for a given data block at t_i, the corresponding feature space F_i will be updated according to the recent data block. The corresponding classifier training/re-training at t_i will be based on F_i.

Types of Features. Malware detection models use different types of features. For the purpose of our analysis, we categorize features into two types: (1) Data-dependent features—new features can be introduced as new data arrives. For example, for PE malware, such data-dependent features can be the set of *imports*, *exports*, *libraries*, and *dll* (carried in new malware samples). (2) Data-independent features—this type of feature is universally applicable to all samples. For example, features such as *file size* and *number of sections* do not need to be introduced by new data and can be computed for any PE files. For our analysis, feature-space drift is mainly considering data-dependent features.

4. Analysis: Android Malware

Dataset. We construct an Android malware dataset sampled from AndroZoo [24] between January 2015 and December 2021. We take the 2015–2016 data from [25] and 2017–2018 data from [9]. Then to obtain more recent data, we sampled APKs from AndroZoo between 2019–2021 to combine with existing datasets. We follow the recommendation from [22] to maintain the temporal and spatial consistency for the evaluation. For each month, we have at least 2,000 samples and maintain a malicious-to-benign ratio of 1:9. Following prior works [22, 26], an APK is labeled "benign" if no VirusTotal engine has flagged it as malicious; an APK is labeled "malicious" if at least four



(a) Training data ratio: 100% (b) Training data ratio: 50% (c) Training data ratio: 25%

Figure 2: Testing performance over time. (a) uses all the training data in the sliding window; (b)-(c) uses less training data.

engines have labeled it so. The rest grayware is excluded. In total, the dataset contains about $32,000$ malware samples and $279,000$ benign samples from 2015–2021.

We used Drebin [1] to extract feature vectors from these Android apps. Drebin has 10 feature categories such as *Activities*, *URLs*, and *APIs*. Each feature is treated as a string entity. Given an app, Drebin produces a binary feature vector of values of 1 or 0—"1" means the app contains this feature (e.g., an API), and "0" means the app does not contain this feature. According to our definition in §3, all of the features are "data-dependent" since new features (e.g., an API that never appear before) can be introduced as new data arrives, which are subject to feature-space drift.

We follow a common approach [27] to performing feature selection before training a detector (to improve training efficiency). We use the LinearSVM L_2 regularizer to select the top 20K features and train an MLP binary classifier with one hidden layer of 1,024 neurons and a dropout rate of 0.2. We use an MLP as the default model because it is a commonly used model for this problem [4, 28, 29, 30]. We will also test SVM and SecSVM [27] for comparison.

Experimental Setup. We divide the data between 2015 and 2021 into 84 months and construct three settings to measure feature-space and data-space drifts.

First, we construct a *baseline* to measure the level of concept drift when the model is never updated after training. We use the first three months of data for training and then test it in the remaining 81 months without any re-training.

Second, we construct a *fixed feature space* setting. The idea is to always keep the same feature set so that we can exclusively measure the data-space drift. We extract the features during the first three months and then fix this feature set for the entire experiment. It means no new features will be added/removed during the subsequential model training and testing. Here, we re-train this model every month using the data in a sliding window of the past three months. After re-training, we test the updated model in the next month. For example, at the end of month 3, we will use data from months 1, 2, and 3 as the training set to retrain the model. We then test this model in month 4 to report the F1 score. This ensures there is no data leakage between training and testing. This process repeats at the end of month 4. Note that we assume abundant labels are available to study the upper-bound performance. In practice, there are various techniques [7, 8, 9, 10] to use a smaller number of labeled samples for retraining, which is not the focus of this paper.

Third, we construct an *updating feature space* setting to examine the impact of the feature-space drift. This setting is similar to the second setting above, except new features from the recent sliding window can be included for retraining. During each re-training, feature updating is done by using a LinearSVM L_2 regularizer to select the top 20K features based on the data in the sliding window. As discussed in §2, feature-space drift always affects the data space because data distribution $D(F)$ is dependent on the feature set F.

Drift in the Android Dataset. As shown in Fig. 2a, the baseline (red line) confirms the existence of concept drift as the model performance (F1 score) is decreasing over time. To counter this effect, we show that re-training helps (both blue and green lines) as the model performance can be maintained at a high level after re-training.

Surprisingly, the blue line and the green line almost overlapped. Recall that the blue line represents the setting where features are updated monthly using recent data, and the green line represents the setting where the features are fixed. This result indicates the model performance is not benefiting from the feature-space update. The fixed feature set (20K features) initially selected in the first three months of 2015 still work well later in 2021. This also indicates that data-space drift (over existing features) is the dominating cause of model performance degradation over time.

To better interpret the result, we revisit the toy example in Fig. 1 (§1). The result means introducing the new feature f_3 (*crypto-lib-X-is-present*) to the feature set is not as critical as updating the data distribution over existing features (f_1 and f_2). A possible reason is that the behavior shift can be readily captured by existing features, e.g., ransomware now does 10x more writes than before, which can be captured by the existing f_1 (*number-of-writes*) during re-training.

We also observe the green line gets a bit higher than the blue line during the later years. A possible explanation is that frequently updating feature space may take in features

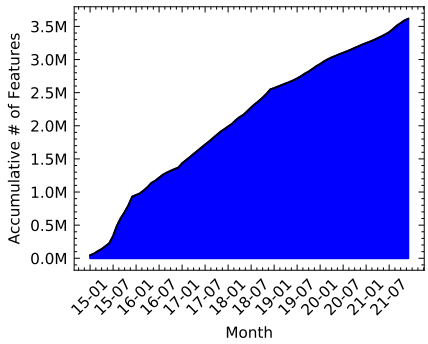


Figure 3: **Cumulative # of features**—This analysis considers all the features that appear in the dataset, before running any feature selection.

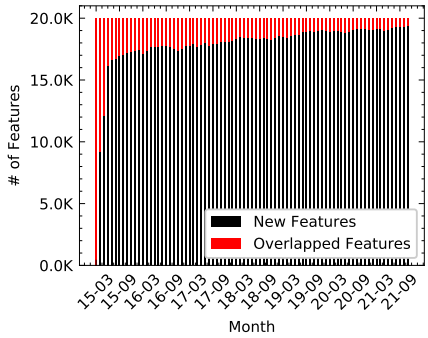


Figure 4: **Updating feature setting**—Number of overlapped features in each training window compared with the initial feature set (first 3 months).

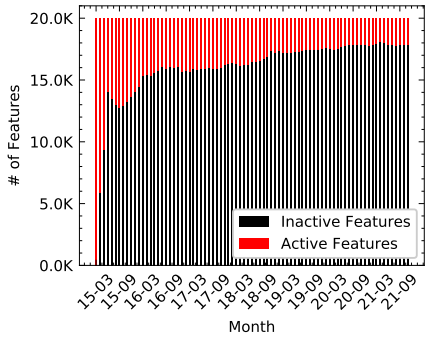
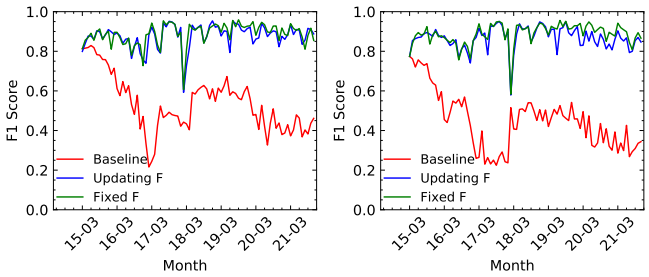


Figure 5: **Fixed feature setting**—Number of active and inactive features in each sliding window. "Active" means at least one sample has this feature.



(a) Feature space size: 10,000 (b) Feature space size: 50,000

Figure 6: Impact of the number of selected features.

that are only temporally useful (i.e., due to short-term drift), but can hurt the models' long-term performance.

Impact of Training Data and Feature Space Size. We further explore under which condition will feature-space drift starts to have an impact. In Fig. 2b and 2c, we first try to reduce the data used for training and re-training. The rationale is that, with less data, it might be more difficult to capture concept drift, and thus the model is more dependent on a good/updated feature space to perform well. This also mimics a real-world scenario where labeled data is limited. The result, however, shows that using less training data does not separate the blue and green lines.

Next, we further explore the impact of feature set size, i.e., the number of selected features. The default feature set size is 20K features. In Fig. 6, we further test 10K and 50K. We find that using a larger or smaller feature set does not make a notable difference.

We have also tested the impact of ML models (using SVM and SecSVM) and also did not observe major differences between the blue and green lines. Due to the space limitation, the result is presented in Appendix B.

Feature Space Analysis. To understand the reasons behind our observation, we further analyze the feature space. A natural question is, is it because there are no major feature changes over time? Fig. 3 suggests the answer is "no". If we consider all the features (without feature selection), there are about 100K unique features observed during the first three months, and 3.6 million features in the end of 2021—a significant number of features are introduced over time.

Then after feature selection, the total number of features is capped at 20K, and thus the level of feature dynamics is reduced. For the "updating feature" setting, Fig. 4 shows the number of features in each sliding window that are overlapped with the first three months (i.e., the initial feature set). The black bar (new features) gets bigger while the red bar (old features) gets smaller over time. This shows new features are indeed selected during periodical retraining.

We further examine the features under the "fixed feature" setting in Fig. 5. For each sliding window, we show the old features that are still active (i.e., appeared in at least one sample in the window) and features that are inactive (i.e., did not appear in any samples). The result indicates that about 2.1K features remain active at the end of 2021. We further analyze the importance score (produced by the feature selection method) and confirm these features have a higher average score (0.054, STD=0.075) than the inactive features (0.032, STD=0.060). This indicates that a small set of important features (selected in the beginning) are sufficient to maintain the model performance over multiple years, without any feature updating. Additional analysis of feature weights and correlations is in Appendix D.

5. Analysis: PE Malware

We validate our observations with PE malware detectors.

Dataset. We use the EMBER PE dataset [3] that contains over 2 million PE files collected between January 2017 and December 2018. The dataset includes 800K malicious, 700K benign, and 500K unlabeled files. For this experiment, we only use labeled malicious and benign files.

EMBER extracts three types of features from a PE file, namely, numerical, boolean, and string features. Numerical features record the numerical value of certain properties of the file (e.g., *file size*, *number of sections*). Boolean features record the binary value ("yes" or "no") such as *has_debug*

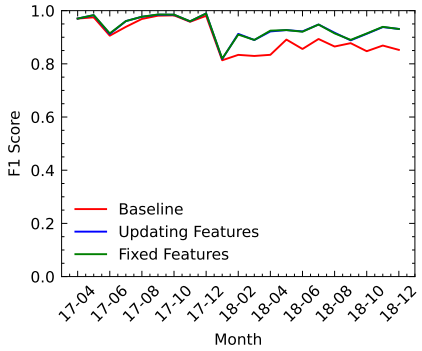


Figure 7: PE results (all features).

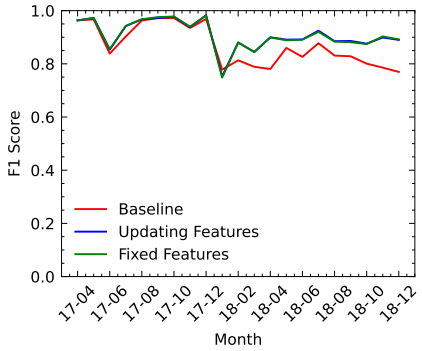


Figure 8: PE results (string features).

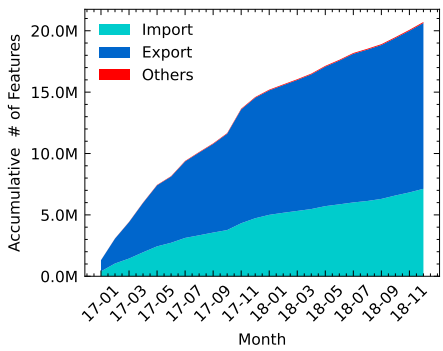


Figure 9: # of unique features over time.

and *has_signature*. String features encode discrete entities such as the set of *imports*, *exports*, and *libraries* in the PE files. There are 10 types of such strings (see Appendix A for the full list). According to our definition in §3, only the string features are "data-dependent" since new features (e.g., libraries that never appear before) can be introduced as new data arrives. Numerical and boolean features are pre-defined (which are not introduced by the new data).

For string features, EMBER proposes to use locality-sensitive hashing (LSH) to convert/embed a set of string entities into a fixed-length vector. For this experiment, we use MinHash [31] for feature embedding which preserves the similarity between the string sets in the embedding space. More details are in Appendix A.

Experimental Setup. We follow the original EMBER paper [3] and use LightGBM to train the malware detector for its high efficiency and good detection performance [32]. We divide the data between 2017 and 2018 into 24 months and test the three settings described in §4: (1) baseline setting, (2) fixed feature space, and (3) updating feature space. Note that settings (2) and (3) are only applied to the *string features* since these features are data-dependent (i.e., subject to feature-space changes).

Drift in the PE Dataset. As shown in Fig. 7, the baseline (red line) confirms that concept drift exists, given the clear drop in the F1 score after the first month of 2018 (dropped by 17%) and stays at the lower level. This result echoes the EMBER paper [3] as the authors intentionally included "harder-to-detect" malware in the 2018 data.

More importantly, we again observe that the blue and green lines are largely overlapping, indicating the two settings have similar results. This suggests data-space drift is the primary cause to model degradation while feature space updating has little impact.

Models with String Features Only. An alternative explanation might be that the features subject to drift (i.e., string features) did not play a major role in the model. To eliminate this possibility, we perform the same experiment by using *string features only*.

As shown in Fig. 8, when only using string features, the conclusion remains the same. The overall performance drops slightly compared to using full features (Fig. 7). However, the blue and green lines still overlapped.

Feature Space Analysis. To show there is indeed a high level of feature dynamics, we plot Fig. 9, which shows the accumulative number of raw string features (before hashing) over time. We note that *import* and *export* contributed to the majority of the unique strings. We can also observe a sudden increase in the number of new features in early 2018, which explains the sudden performance drop during that time (Fig. 8). The result confirms that a large number of new features (strings) are introduced, from 4.4 million in the first three months to 20.7 million by the end of 2018.

Despite the high level of feature dynamics, the classifier's performance is not affected, possibly due to feature embedding. The hashing function (LSH) has helped to map a large, sparse feature space (i.e., 20.71 million strings) into a fixed-sized dense feature space (in our case, the hash vector length is 2,381), which can stabilize the feature space. Another reason is that the initial feature set (first three months) is still quite actively used by samples in the last month of the two-year period (see an example in Fig. 10). These features remain effective in separating malicious from benign samples (F1=0.93, Fig. 7).

6. Online Learning

Finally, we revisit systems that proactively perform feature space updating via *online learning*. We pick DroidEvolver [11] considering it is more advanced than the earlier variants (e.g., [13]). A recent paper introduced improvements on DroidEvolver and released DroidEvolver++ [12]. Due to the space limit, we report the DroidEvolver analysis in this section. The experiment on DroidEvolver++ has a similar conclusion, which is shown in Appendix C.

DroidEvolver. DroidEvolver uses online learning to continuously update the model as new data arrives. Unlike our experiments in §4 and §5 (models are updated each month with *batches of samples*), DroidEvolver updates its model incrementally with *each individual sample* as it arrives (i.e., stream-based). DroidEvolver trains an ensemble of 5 linear online learning models. At the testing time, DroidEvolver produces a prediction for a sample based on a weighted sum of decision scores from the 5 models. To handle concept drift, DroidEvolver maintains an *app buffer* to store a small set of samples that represent the current data distribution.

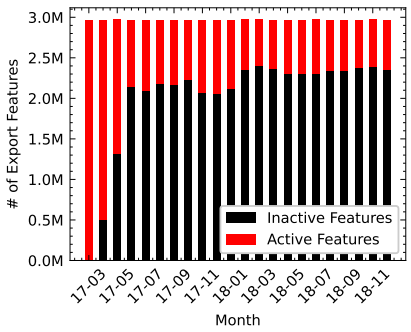
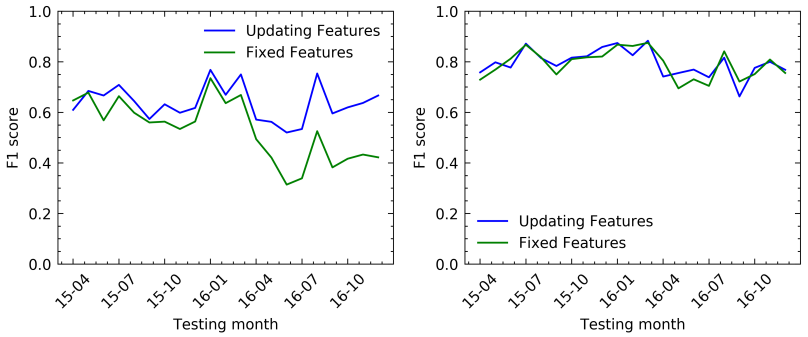


Figure 10: "Fixed feature" setting: we show the number of active and inactive *export* features in each sliding window.



(a) Pseudo Labels (b) Ground Truth Labels

Figure 11: DroidEvolver performance over time.

The buffer is used to identify new samples that deviate from the current distribution and also flag "aging models" that start to give less accurate predictions. The aging model then will be updated using the new sample. To get the labels for the model updating, DroidEvolver produces a "pseudo-label" for the new sample based on the ensemble prediction of "non-aging" models. Importantly, during model updating, the feature set is also extended to include any previously unseen features present in the new sample.

Experimental Setup. This experiment is not to diminish the value of DroidEvolver, which is one of the first papers to study stream-based model updating for malware detectors. Our goal is to obtain a deeper understanding of the impact of feature-space and data-space drifts. The original DroidEvolver is evaluated on an Android malware dataset with API features. Without access to the original data, we use Drebin features (which include APIs as features).

DroidEvolver updates the model based on each new sample (not in batches), and thus it takes a long time to run. For this test, we focus on 2015–2016 data (24 months). We train the initial model ensemble with the first three months of data in 2015. After that, we run DroidEvolver to perform malware detection and model updating as new samples arrive (with 1,000 random samples each month). We aggregate the testing results per month to report F1. To examine the impact of feature updating, we create two versions, one with feature updating enabled (DroidEvolver's original design), and one never updates the feature space.

As mentioned, DroidEvolver does not need any "ground-truth" labels for the model updating—they rely on pseudo-labels voted by the model ensemble. However, recent works pointed out that model updating with its own predictions (pseudo-labels) can lead to self-poisoning [12, 33]. As such, we also test the case where model updating is done with ground-truth labels for DroidEvolver.

Results. Fig. 11a shows the performance using the pseudo labels. It has a much lower F1 score (around 0.6–0.7), compared with updating with ground-truth labels (F1 over 0.8, Fig. 11b). This confirms the concerns from prior work that pseudo labels can cause self-poisoning [12, 33] which started as early as the first testing month (15-04).

Interestingly, when using ground-truth labels (Fig. 11b), there is almost no difference between updating and fixing the feature space. This is consistent with our prior observations. However, when using "pseudo labels", we observe that updating feature space has helped to improve the F1 score. Recall that pseudo-labels are noisy and often contain incorrect labels. This result indicates that feature updating might be helpful when label quality is low.

7. Discussion and Conclusion

In this paper, we decouple feature-space drift from data-space drift to examine their impact on malware detectors. Empirically, we find that data-space drift is the dominating contributor to model performance degradation over time while feature-space updating has little to no impact. This is consistently observed across different feature types and feature engineering methods, across Android and PE malware detectors. The takeaway is that a well-constructed feature set can effectively support model re-training (to catch up with data-space drift) without updating the feature set itself. This is true despite a large number of new features being introduced by incoming new samples.

Does it mean feature updating is completely unnecessary? We don't think that's the case either. For example, in §6, we show that feature-space updating seems to help when the labels used for model retraining are noisy (less accurate). Future work is needed to further investigate this condition to draw a more reliable conclusion.

On the flip side, what's the benefit of not updating features frequently? First, during model re-training, if a brand new feature set is used, it means the model will need to be either re-trained from scratch on the new feature space (costly) or use simpler *linear models* (less accurate) for incremental updating like DroidEvolver. In comparison, with a fixed feature set, there is more flexibility in model choices and incremental updating methods. Second, frequent feature updating may take on features that are only temporally useful (i.e., due to short-term drift), but hurt the models' long-term performance (e.g., Fig. 2). We believe more work is needed to understand the impact of feature-space updating.

Acknowledgment. This work was supported in part by NSF grants CNS-2055233 and CNS-1955719, C3.AI Research, IBM-Illinois Discovery Accelerator Institute, and a gift from AVAST.

References

[1] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *Proc. of NDSS*, 2014.

[2] M. Lindorfer, M. Neugschwandtner, and C. Platzer, "Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis," in *Prof. of COMPSAC*, 2015.

[3] H. S. Anderson and P. Roth, "Ember: an open dataset for training static pe malware machine learning models," *arXiv preprint arXiv:1804.04637*, 2018.

[4] Y. Chen, S. Wang, D. She, and S. Jana, "On training robust PDF malware classifiers," in *Proc. of USENIX Security*, 2020.

[5] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck, "Dos and don'ts of machine learning in computer security," in *Proc. of USENIX Security*, 2022.

[6] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM computing surveys (CSUR)*, 2014.

[7] L. Yang, W. Guo, Q. Hao, A. Ciptadi, A. Ahmadzadeh, X. Xing, and G. Wang, "CADE: Detecting and explaining concept drift samples for security applications," in *Proc. of USENIX Security*, 2021.

[8] D. Han, Z. Wang, W. Chen, Y. Zhong, S. Wang, H. Zhang, J. Yang, X. Shi, and X. Yin, "Deepaid: Interpreting and improving deep learning-based anomaly detection in security applications," in *Proc. of CCS*, 2021.

[9] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro, "Transcend: Detecting concept drift in malware classification models," in *Proc. of USENIX Security*, 2017.

[10] F. Barbero, F. Pendlebury, F. Pierazzi, and L. Cavallaro, "Transcending transcend: Revisiting malware classification in the presence of concept drift," in *Proc. of IEEE S&P*, 2022.

[11] K. Xu, Y. Li, R. Deng, K. Chen, and J. Xu, "Droidevolver: Self-evolving android malware detection system," in *Proc. of Euro S&P*, 2019.

[12] Z. Kan, F. Pendlebury, F. Pierazzi, and L. Cavallaro, "Investigating labelless drift adaptation for malware detection," in *Proc. of AISec*, 2021.

[13] A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu, "Context-aware, adaptive, and scalable android malware detection through online learning," *IEEE TETCI*, 2017.

[14] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *Proc. of IEEE S&P*, 2010.

[15] M. Baena-Garcıa, J. del Campo-Ávila, R. Fidalgo, A. Bifet, R. Gavalda, and R. Morales-Bueno, "Early drift detection method," in *Proc. of Workshop on knowledge discovery from data streams*, 2006.

[16] A. Bifet and R. Gavalda, "Learning from time-changing data with adaptive windowing," in *Proc. of SDM*, 2007.

[17] M. Harel, S. Mannor, R. El-Yaniv, and K. Crammer, "Concept drift detection through resampling," in *Proc. of ICML*, 2014.

[18] D. M. dos Reis, P. Flach, S. Matwin, and G. Batista, "Fast unsupervised online drift detection using incremental kolmogorov-smirnov test," in *Proc. of KDD*, 2016.

[19] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto, "Novel feature extraction, selection and fusion for effective malware family classification," in *Proc. of CODASPY*, 2016.

[20] T. Chakraborty, F. Pierazzi, and V. Subrahmanian, "Ec2: Ensemble clustering and classification for predicting android malware families," *IEEE TDSC*, 2017.

[21] A. Kantchelian, S. Afroz, L. Huang, A. C. Islam, B. Miller, M. C. Tschantz, R. Greenstadt, A. D. Joseph, and J. D. Tygar, "Approaches to adversarial drift," in *Proc. of AISec*, 2013.

[22] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "TESSERACT: Eliminating experimental bias in malware classification across space and time," in *Proc. of USENIX Security*, 2019.

[23] X. Zhang, Y. Zhang, M. Zhong, D. Ding, Y. Cao, Y. Zhang, M. Zhang, and M. Yang, "Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware," in *Proc. of CCS*, 2020.

[24] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proc. of MSR*, 2016.

[25] L. Yang, Z. Chen, J. Cortellazzi, F. Pendlebury, K. Tu, F. Pierazzi, L. Cavallaro, and G. Wang, "Jigsaw puzzle: Selective backdoor attack to subvert malware classifiers," *arXiv preprint arXiv:2202.05470*, 2022.

[26] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro, "Intriguing properties of adversarial ML attacks in the problem space," in *Proc. of IEEE S&P*, 2020.

[27] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli, "Yes, machine learning can be more secure! a case study on android malware detection," *IEEE TDSC*, 2017.

[28] K. Xu, Y. Li, R. H. Deng, and K. Chen, "Deeprefiner: Multi-layer android malware detection system applying deep neural networks," in *Proc. of Euro S&P*, 2018.

[29] G. Severi, J. Meyer, S. Coull, and A. Oprea, "Explanation-guided backdoor poisoning attacks against malware classifiers," in *Proc. of USENIX Security*, 2021.

[30] H. Li, S. Zhou, W. Yuan, X. Luo, C. Gao, and S. Chen, "Robust android malware detection against adversarial example attacks," in *Proc. of WWW*, 2021.

[31] W. Wu, B. Li, L. Chen, J. Gao, and C. Zhang, "A review for weighted minhash algorithms," *IEEE TKDE*, vol. 34, no. 6, pp. 2553–2573, 2022.

[32] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," in *Proc. of NeurIPS*, 2017.

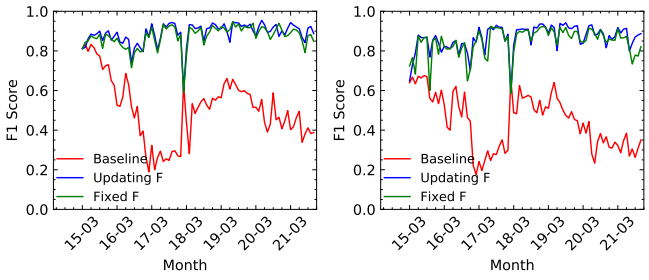
[33] G. Andresini, F. Pendlebury, F. Pierazzi, C. Loglisci, A. Appice, and L. Cavallaro, "Insomnia: Towards concept-drift robustness in network intrusion detection," in *Proc. of AISec*, 2021.

[34] M. A. Hall, "Correlation-based feature selection for machine learning," Ph.D. dissertation, The University of Waikato, 1999.

Appendix A.
EMBER: Embedding String-based Features

In the EMBER dataset [3], string-based features are the main source of feature-space drift, and we provide more context for their embedding method. There are ten categories of string features including *section_name*, *section_characteristics*, *library*, *import*, *export*, *machine*, *subsystem*, *dll_characteristics*, *header_characteristics*, and *magic*. For each file, the raw features of each category are presented as a set of discrete strings/entities. Given the large number of unique entities, we follow EMBER [3] to embed the string features using a hashing function that maps a set of strings to a fixed-length vector. We choose the vector length for each string category based on the suggestion of the original paper [3]. We use a locality-sensitive hash (LSH) method—the advantage is that LSH preserves the similarity between sets, i.e., if two sets are more similar (i.e., with a bigger intersection), their hashed vectors also have a smaller distance in the embedding space. Hashing helps to map a sparse feature space with potentially tens of millions of features into fixed-sized dense vectors (e.g., size of 2,381). This helps to improve the efficiency of training. Note that the original EMBER paper [3] used FeatureHasher in the sklearn library which is an LSH with respect to cosine distance. We used MinHash [31] for easy implementation, and we confirmed that both hashing algorithms have equally good performances (i.e., $< 1\%$ difference in F1 score).

Appendix B.
Impact of Different Models



(a) Model: SVM (b) Model: SecSVM

Figure 12: Impact of different types of models.

In this section, we evaluate the impact of model choices on our experiment result on the Android dataset. In addition to the MLP model (used in §4), prior works have also used LinearSVM and SecSVM for Android malware detection [26]. Note that SecSVM [27] is a variant of the SVM model designed to be more robust against evasion attacks. The main idea of SecSVM is to maintain more evenly-distributed feature weights such that the model cannot be easily evaded by manipulating a few features. We follow the same experimental methodology from §4, and test LinearSVM and SecSVM respectively under the default setting (20K feature set size, 100% training data). The results are presented in Figure 12. We find that our conclusion remains the same. With LinearSVM and SecSVM, the blue and green lines are still largely overlapping, indicating that data-space drift is still the dominating contributor to concept drift.

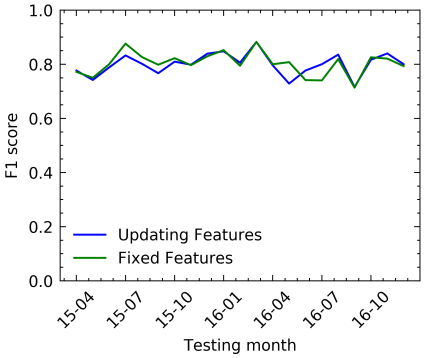


Figure 13: DroidEvolver++ performance over time.

Appendix C.
Online Learning with DroidEvolver++

A recent paper introduced a series of improvements on DroidEvolver and released DroidEvolver++ [12]. These improvements include two main categories: (1) a new pseudo-label generation method (to reduce self-poisoning), and (2) other small design improvements such as modifying the voting scheme for the model ensemble and the updating scheme for "app buffer". To validate our observation on DroidEvolver++, we run the same experiment described §6. Here, to avoid the distraction of the self-poisoning problem from pseudo-labels, we use ground-truth labels for the test (other design improvements from DroidEvolver++ are applied). The result is reported in Fig. 13. We show that the blue line (updating features) and the green line (using fixed features) are still very close, confirming the conclusion remains consistent, namely, feature updating has little impact on the model performance over time.

Appendix D.
Feature Correlation and Weight Analysis

In this section, we provide additional analysis to understand the correlation between features in the fixed feature space and those in the updated feature space, to understand the impact of feature updating.

In §4, we observe that updating the feature space has little impact on improving the model performance. Other than the explanation provided in §4, another possible explanation is that the newly added features are strongly correlated with the replaced/removed features, and thus making the update had little impact on the model.

To validate this hypothesis, we run a quick experiment on the last 3-month window of the Android dataset (ending in 2021-11, see Fig. 4). For this time window, we divide the features into three groups, by comparing features in the fixed-feature setting and the updating-feature setting: (1) 644 features in both feature sets (overlapped features); (2) 19,356 features newly selected under the updating setting (new features); and (3) 1,495 features that exist in the fixed feature space and the last 3-month time window, but are removed from the updated feature set (removed features). We then calculate the average Pearson's correlation coefficients [34] between the above three feature groups. The correlation coefficient is 0.0114 between "overlapped" and "new" features, 0.0259 between the "overlapped" and "removed" features, and 0.0024 for the "new" and "removed" features. The result indicates that the newly added features are least correlated with the removed features, which *does not support the above hypothesis*. In other words, the ineffectiveness of feature updating is not caused by strong correlations between the new and removed features.

Another possible explanation for the lack of impact of feature updating is that the new features do not have significantly higher weights than those of the removed features. Here, using the above experiment setup, we further calculate the average feature weights for the "new" feature set and the "removed" feature set, based on the last 3-month time window. We confirm that the newly selected features indeed have much higher weights (AVG=0.0104, STD=0.010) than those of removed features (AVG=0.0008, STD=0.001).

Overall, the results help to eliminate these alternative explanations for the observation in §4.