Bucks for Buckets (B4B): Active Defenses Against Stealing Encoders

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

Machine Learning as a Service (MLaaS) APIs provide ready-to-use and high-utility encoders that generate vector representations for given inputs. Since these encoders are very costly to train, they become lucrative targets for model stealing attacks during which an adversary leverages query access to the API to replicate the encoder locally at a fraction of the original training costs. We propose *Bucks for Buckets (B4B)*, the first *active defense* that prevents stealing while the attack is happening without degrading representation quality for legitimate API users. Our defense relies on the observation that the representations returned to adversaries who try to steal the encoder's functionality cover a significantly larger fraction of the embedding space than representations of legitimate users who utilize the encoder to solve a particular downstream task. B4B leverages this to adaptively adjust the utility of the returned representations according to a user's coverage of the embedding space. To prevent adaptive adversaries from eluding our defense by simply creating multiple user accounts (sybils), B4B also individually transforms each user's representations. This prevents the adversary from directly aggregating representations over multiple accounts to create their stolen encoder copy. Our active defense opens a new path towards securely sharing and democratizing encoders over public APIs.

1 Introduction

In model stealing attacks, adversaries extract a machine learning model exposed via a public API by repeatedly querying it and updating their own stolen copy based on the obtained responses. Model stealing was shown to be one of the main threats to the security of machine learning models in practice [36]. Also in research, since the introduction of the first extraction attack against classifiers [38], a lot of work on improving stealing [26, 32, 38, 39], extending it to different model types [8, 35], and proposing adequate defenses [17, 24, 25, 30] has been put forward. With the recent shift in learning paradigms from supervised to self supervised learning (SSL), especially the need for new defenses becomes increasingly pressing. From an academic viewpoint, the urge arises because it was shown that SSL models (*encoders*) are even more vulnerable to model stealing [15, 28, 34] than their supervised counterparts. This is because whereas supervised models' output is low dimensional, *e.g.,* per-class probabilities or pure labels, SSL encoders output high-dimensional representation vectors that encode a larger amount of information and thereby facilitate stealing. In addition, from a practical industry's viewpoint, defenses are required since many popular API providers, such as Cohere, OpenAI, or Clarify [1–3] already expose their high-value SSL encoders via APIs to a broad range of users.

Most of the current defenses against encoder stealing are *reactive*, *i.e.,* they do not actively prevent the stealing but rather aim at detecting it by adding watermarks to the encoder [13, 15] or performing dataset inference to identify stolen copies [16]. Since at the point of detection, the damage of stealing has already been inflicted, we argue that reactive defenses intervene too late and we advocate for

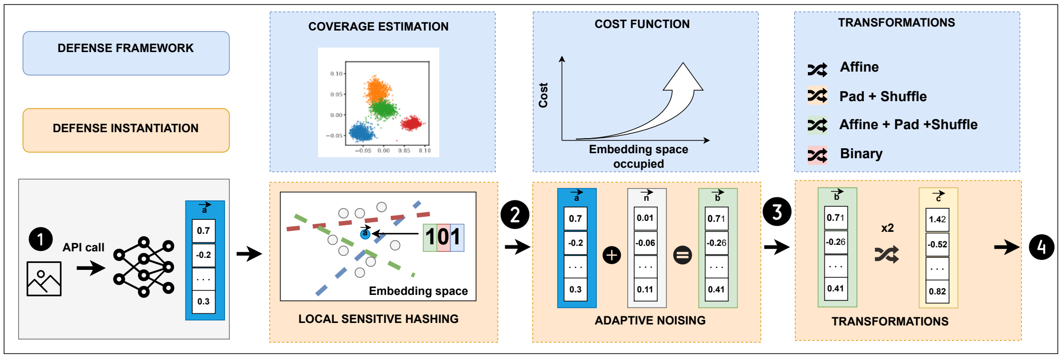


Figure 1: **Overview of B4B.** In the upper part, we present our B4B framework that consists of three modular building blocks: (1) A coverage estimation to track the fraction of embedding space covered by the representations returned to each user, (2) a cost function that serves to map the coverage to a concrete penalty to prevent stealing, and (3) per-user transformations that are applied to the returned representations to prevent sybil attacks. In the lower part, we present a concrete instantiation of B4B and the operation flow of our defense: ① The API calculates representations for the incoming queries. ② We instantiate the coverage estimation with local sensitive hashing and estimate the covered space as the fraction of *hash buckets* occupied. We calibrate the costs by adding noise to the representations according to the coverage. ③ We apply a set of transformations on a per-user basis. ④ The noised and transformed representations are returned to the user.

active defenses that prevent stealing while it is happening. Yet, active defenses are challenging to implement because they not only need to prevent stealing but also should preserve the utility of representations for legitimate users. The only existing active defense against encoder stealing [28] falls short on this latter aspect since it significantly degrades the quality of representations for all users.

To close the gap between required and existing defenses, we propose *Bucks for Buckets (B4B)*, the first active defense against encoder stealing that does not harm utility for legitimate users. B4B leverages the observation that the representations returned to adversaries who try to steal the encoder's functionality cover a significantly larger fraction of the full embedding space than representations of legitimate users who utilize the encoder to solve a particular downstream task. To turn this observation into a practical defense, B4B is equipped with three modular building blocks: (1) The first building block is a tracking mechanism that continuously estimates the fraction of the embedding space covered by the representations returned to each user. The intuition why this is relevant is that by covering large fractions of the embedding space, the representations will suffice for an adversary to reproduce the encoder's functionality, *i.e.,* to successfully steal it. (2) B4B's second building block consists of a cost function to translate the covered fraction of the embedding space into a concrete penalty. We require this cost function to significantly penalize adversaries trying to steal the model while having only a minimal effect on legitimate users. (3) The third building block contains transformations that can be applied to the representations on a per-user basis to prevent adaptive attackers from circumventing our defense by creating multiple user accounts (sybils) and distributing their queries over these accounts such that they minimize the overall cost. We present the different building blocks of B4B in Figure 1.

While B4B's modularity enables different instantiations of the three building blocks, we propose a concrete end-to-end instantiation to showcase the practicability of our approach. To implement tracking of the covered embedding space, we employ *local sensitive hashing* that maps any representation returned to a given user into a set of hash ***buckets***. We base our cost function (*i.e.,* the ***"bucks"***) on utility and make B4B add noise to the representations with a magnitude that increases with the number of buckets occupied by the given user. While the scale of noise added to legitimate users' representations does not harm their downstream performance due to their small embedding space coverage, the representations returned to an adversary become increasingly noisy—significantly degrading the performance of their stolen encoder. Finally, we rely on a set of transformations (*e.g.,* affine transformations, shuffling, padding) that preserve downstream utility [16]. While, as a consequence, legitimate users remain unaffected by these transformations, adversaries cannot directly combine the representations obtained through different sybil accounts anymore to train their stolen copy of the encoder. Instead, they first have to remap all representations into the same embedding space, which we show causes both query and computation overhead and still reduces the performance of the stolen encoder.

In summary, we make the following contributions:

1. We present B4B, the first active defense against encoder stealing that does not harm legitimate users' downstream performance. B4B's three building blocks enable penalizing adversaries whose returned representations cover large fractions of the embedding space and prevent sybil attacks.
2. We propose a concrete instantiation of B4B that relies on local sensitive hashing and decreases the quality of representations returned to a user once their representations fill too many hash buckets.
3. We provide an end-to-end evaluation of our defense to highlight its effectiveness in offering high utility representations for legitimate users and degrading the performance of stolen encoders in both the single and the sybil-accounts setup.

2 Related Work

Model Extraction Attacks. The goal of the model extraction attacks is to replicate the functionality of a victim model f_v trained on a dataset D_v. An attacker has a black box access to the victim model and uses a stealing dataset $D_s = \{q_i, f_v(q_i)\}_{i=1}^{n}$, consisting of queries q_i and the corresponding outputs $f_v(q_i)$ returned by the victim model, to train a stolen model f_s. Model extraction attacks have been shown against various types of models including classifiers [23, 38] and encoders [15, 34].

Self Supervised Learning and Encoders. SSL is an increasingly popular machine learning paradigm. It trains encoder models to generate representations from complex inputs without relying on explicit labels. These representations encode useful features of a given input, enabling efficient learning for multiple downstream tasks. Many SSL frameworks have been proposed [9–11, 21, 22, 42]. In our work, we focus on the two popular SSL vision encoders, namely SimSiam [11] and DINO [9], which return high-quality representations that achieve state-of-the-art performance on downstream tasks when assessed by training a linear classifier directly on representations. SimSiam trains with two Siamese encoders with directly shared weights. A prediction MLP head is applied to one of the encoders f_1, and the other encoder f_2 has a stop-gradient, where both operations are used for avoiding collapsing solutions. In contrast, DINO shares only architecture (not weights) between a student f_1 and a teacher model f_2, also with the stop-gradient operation, but not the prediction head. While SimSiam uses convolutional neural networks (CNNs), DINO also employs vision transformers (ViTs). Both frameworks use a symmetrized loss of the form $\frac{1}{2}g(f_1(x_1), f_2(x_2)) + \frac{1}{2}g(f_1(x_2), f_2(x_1))$ in their optimization objectives, where $g(\cdot, \cdot)$ is negative cosine similarity for SimSiam and cross-entropy for DINO. SimSiam and DINO's similarities and differences demonstrate our method's broad applicability across SSL frameworks. More details can be found in Appendix E.

Stealing Encoders. The stealing of SSL encoders was shown to be extremely effective [15, 28, 34]. The goal of extracting encoders is to maximize the similarity of the outputs from the stolen local copy and the original representations output by the victim encoder. Therefore, while training the stolen copy, the adversary either imitates a self-supervised training using a contrastive loss function, *e.g.,* InfoNCE [10] or SoftNN [20] or directly matches both models' representations via the Mean Squared Error (MSE) loss. To reduce the number of queries sent to the victim encoder, the attack proposed in [28] leverages the key observation that the victim encoder returns similar representations for any image and its augmented versions. Therefore, a given image can be sent to the victim while the stolen copy is trained using many augmentations of this image, where the representation of a given augmented image is approximated as the one of the original image produced by the victim encoder.

Defending Encoders. Recently, watermarking [7, 24, 40] methods have been proposed to detect stolen encoders [13, 15, 41]. Many of these approaches use downstream tasks to check if a watermark embedded into a victim encoder is present in a suspect encoder. Dataset inference [29] is another type of encoder ownership resolution. It uses the victim's training dataset as a unique signature, leveraging the following observation: for a victim encoder trained on its private data as well as for its stolen copies, the distribution of the representations generated from the victim's training data differs from the distribution of the representations generated on the test data. In contrast, for an independently trained encoder, these two distributions cannot be distinguished, allowing the detection of stolen copies [16]. However, all the previous methods are *reactive* and aim at detecting the stolen encoder instead of *actively* preventing the attack. The only preliminary active defenses for encoders were proposed by [28]. They either perturb or truncate the answers to poison the training objective of an attacker. These operations were shown to harm substantially the performance of legitimate users, which renders the defense impractical. In contrast, our B4B has negligible impact on the quality of representations returned to legitimate users.

3 Actively Defending against Model Stealing with B4B

B4B aims at actively preventing model stealing while preserving high-utility representations for legitimate users. Before introducing the three main building blocks of B4B, namely (1) the estimation of embedding space coverage, (2) the cost function, and (3) the transformation of representations (see Figure 1), we detail our threat model and the observation on embedding space coverage that represents the intuition behind our approach.

3.1 Threat Model and Intuition

Our setup and the resulting threat model are inspired by public APIs, such as Cohere, OpenAI, or Clarify [1–3] that expose encoders to users through a pre-defined interface. These encoders are trained using SSL on large amounts of unlabeled data, often crawled from the internet, and therefore from diverse distributions. We notice that to provide rich representations to multiple users, the training dataset of the encoder needs to be significantly more diverse than the individual downstream tasks that the users query for representations. For instance, if the encoder behind the API is trained on the ImageNet dataset, then the legitimate users are expected to query the API for downstream tasks, such as CIFAR10 or SVHN. Similarly, if the encoder is trained on CIFAR10, the expected downstream tasks are MNIST or Fashion MNIST. Yet, in the design of our defense, we consider adversaries who can query the encoder with arbitrary inputs to obtain high-dimensional representation vectors from the encoder. Our defense is independent of the protected encoder's architecture and does not rely on any assumption about the adversary's data and query strategy.

We argue that even in this restricted setup, our defense can distinguish between adversaries and legitimate users by analyzing the distribution of representations returned to them. In Figure 2, by using PCA to project representations for different datasets to a two-dimensional space, we visualize that representations for different downstream tasks cluster in *disjoint* and *small sub-spaces* of the full embedding space. The representations were obtained from a SimSiam encoder originally trained on ImageNet (we observe similar clustering for DINO shown in Appendix F). As a result, legitimate users can be characterized by their representations' small coverage of the embedding space. In contrast, the adversary does not aim at solving a particular downstream task. They instead would want to obtain representations that cover large fractions of the embedding space. This enables reproducing the overall functionality of the encoder (instead of only learning some local task-specific behavior). Indeed, it has been empirically shown by prior work, such as [15], that stealing with multiple distributions, *e.g.,* by relying on the complex ImageNet dataset, yields higher performance of the stolen encoder on various downstream tasks than stealing with a downstream dataset, such as CIFAR10. As a result, intuitively, we can identify and penalize adversaries based on their coverage of the embedding space, which will be significantly larger than the coverage of legitimate users. We leverage this intuition to build our B4B defense and present our three main building blocks in the following sections.

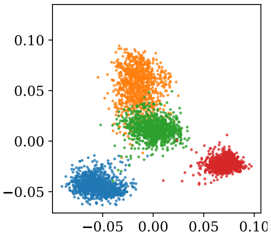


Figure 2: **Representations from Different Tasks Occupy Different Sub-Spaces of the Embedding Space. Presented for FashionMNIST, SVHN, CIFAR10, and STL10.**

3.2 Building Block 1: Coverage Estimation of the Embedding Space

The first building block of our B4B serves to estimate and continuously keep track of the fraction of the embedding space occupied by any given user. Let $\mathcal{E}$ denote our embedding space of dimension s, further, let U be a user with a query dataset $D = q_1, \ldots, q_n \in \mathcal{D}$ and let $f_v : \mathcal{D} \to \mathcal{E}$ be our protected victim encoder that maps data points from the input to the embedding space. Assume user U has, so far, queried a subset of their data points $q_1, \ldots, q_j$ with $j \leq n$ to the encoder and obtained the representations $r_1, \dots, r_j$ with each $r_i \in \mathbb{R}^s$. We aim to estimate the true fraction of the embedding space $\mathcal{E}_f^U$ that is covered by all returned representations $r_1, \dots, r_j$ to user U and denote our estimate by $\tilde{\mathcal{E}}_f^U$.

Local Sensitive Hashing. One of the methods to approximate the occupied space by representations returned to a given user is via Local Sensitive Hashing (LSH) [37]. We rely on this approach for the concrete instantiation of our B4B and use it to track per-user coverage of the embedding space. Standard (cryptographic) hash functions are characterized by high dispersion such that hash collisions are minimized. In contrast, LSH hashes similar data points into the same or proximal, so-called *hash buckets*. This functionality is desired when dealing with searches in high-dimensional spaces or with a large number of data points. Formally, an LSH function $\mathcal{H}$ is defined for a metric space $\mathcal{M} = (M, d)$, where d is a distance metric in space M, with a given threshold $T > 0$, approximation factors $f > 1$, and probabilities P_1 and P_2, where $P_1 \gg P_2$. $\mathcal{H}$ maps elements of the metric space to buckets $b \in B$ and satisfies the following conditions for any two points $q_1, q_2 \in M$: (1) If $d(q_1, q_2) \leq T$, then $\mathcal{H}(q_1) = \mathcal{H}(q_2)$ (*i.e.,* q_1 and q_2 collide in the same bucket b) with probability at least P_1. (2) If $d(q_1, q_2) \geq fT$, then $\mathcal{H}(q_1) = \mathcal{H}(q_2)$ with probability at most P_2.

3.3 Building Block 2: Cost Function Design

Once we can estimate the coverage of an embedding space for a given user U as $\tilde{\mathcal{E}}_f^U$, we need to design a cost function $\mathcal{C} : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ that maps from the estimated coverage to a cost. The cost function needs to be designed such that it does not significantly penalize legitimate users while imposing a severe penalty on adversaries to effectively prevent the encoder from being stolen. The semantics of the cost function's range depend on the type of costs that the defender wants to enforce. We discuss a broad range of options in Appendix C. These include monetary cost functions to adaptively charge users on a batch-query basis depending on their current coverage, costs in the form of additional computation that users need to perform in order to obtain their representations, similar to the proof of work in [17], costs in the form of delay in the interaction with the encoder behind the API [4], or costs in form of disk space that needs to be reserved by the user (similar to a proof of space [18, 19]). Which type of cost function is most adequate depends on the defender's objective and setup.

Exponential Cost Functions to Adjust Utility of Representations. In the concrete instantiation of B4B that we present in this work, we rely on costs in the form of the utility of the returned representations. We choose this concrete instantiation because it is intuitive, effective, and can be directly experimentally assessed. Moreover, it is even suitable for public APIs where, for example, no monetary costs are applicable. We adjust utility by adding Gaussian noise with different standard deviation σ to the returned representations. Since we do not want to penalize legitimate users with small coverage but make stealing for adversaries with growing coverage increasingly prohibitive, we instantiate an exponential cost function that maps from the fraction of hash buckets occupied by the user to a value for σ. We choose the general form of this function as

$$f_{\lambda,\alpha,\beta}(\tilde{\mathcal{E}}_f^U) = \lambda \times (\exp^{\ln \frac{\alpha}{\lambda} \times \tilde{\mathcal{E}}_f^U \times \beta^{-1}} - 1) \tag{1}$$

where $\lambda < 1$ compresses the curve of the function to obtain low function values for small fractions of occupied buckets, and then we set a target penalty α for our cost function at a specified fraction of filled buckets β. For instance, if we want to enforce a σ of 5 at 90% of filled buckets (*i.e.,* for $\tilde{\mathcal{E}}_f^U = 0.9$), we would need to set $\alpha = 5$ and $\beta = 0.9$.

3.4 Building Block 3: Per-User Representation Transformations against Sybil Attacks

Given that our defense discourages users from querying with a wide variety of data points from different distributions, an adversary could create multiple fake user accounts (sybils) and query different data subsets with more uniform representations from each of these accounts. By combining all the obtained representations and using them to train a stolen copy, the adversary could overcome the increased costs of stealing. To defend against such sybil attacks, we propose individually transforming the representations on a per-user level before returning them. As a result, the adversary would first have to map all the representations to one single unified space before being able to jointly leverage the representations from different accounts for their stolen copy. Formally, for a given query

q_i, the protected victim encoder produces a representation $r_i = f_v(q_i)$, which is transformed by a user-specific transformation $T_U(r_i)$ before being returned to the querying user U. We formulate two concrete requirements for the transformations. First, they should preserve utility for legitimate users on their downstream tasks, and second, they should be costly to reverse for an adversary.

Utility Preserving Transformations. As a concrete instantiation for our B4B, we propose a set of transformations that fulfill the above-mentioned two requirements: (1) *Affine* where we apply affine transformations to representations, (2) *Pad* where we pad representations with constant values, (3) *Add* where we add constant values at random positions within representations, (4) *Shuffle* where we shuffle the elements in the representation vectors, and (5) *Binary* where the original representations are mapped to binary vectors relying on a random partitioning of the representation space. To preserve the full amount of information contained in the original representations, in our binary transformations, we tune the length of binary representations. We visualize the operation of each of these transformations in Appendix C. All these transformations can additionally be combined with each other, which further increases the possible set of transformations applied per user. This renders it impossible for an adversary to correctly guess and reverse the applied representation. Instead, the adversary has to remap the representations over all accounts into a single embedding space in order to unify them and leverage them for training of their stolen encoder copy. We present an exhaustive list of strategies that adversaries can apply for the remapping in Appendix D. All the attack methods reduce to the minimum of remapping between representations of a pair of users, *i.e.,* they are at least as complex as mapping between two separate accounts. In the next section, we show that our defense already impedes stealing for an adversary with two accounts.

4 Empirical Evaluation

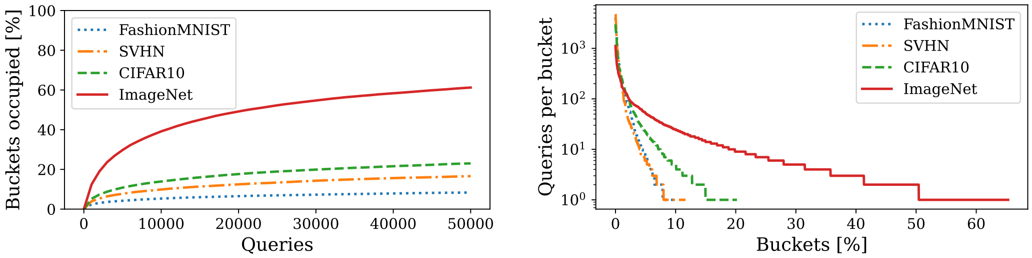
We first empirically evaluate our instantiation of B4B's three building blocks and show how to calibrate each of them for our defense. Finally, we provide an end-to-end evaluation that highlights B4B's effectiveness in preserving downstream utility for legitimate users while successfully preventing the stealing by adversaries.

Experimental Setup. We conduct experiments on various kinds of downstream tasks and two popular SSL encoders. To test our defense, we use FashionMNIST, SVHN, STL10, and CIFAR10 as our downstream datasets, each with standard train and test splits. For stealing, we utilize training data from ImageNet. We rely on encoder models from the SimSiam [11] and the DINO [9] SSL frameworks. As our victim encoders, we use the publicly available ResNet50 model from SimSiam trained for 100 epochs on ImageNet and the ViT Small DINO encoder trained for 800 epochs on ImageNet, each using batch size 256. The ViT architecture takes as input a grid of non-overlapping contiguous image patches of resolution NxN. In this paper, we typically use $N = 16$. The Simsiam encoder has an output representation dimension of 2048, while DINO returns 1536 dimensional representations. We examine the utility of downstream classifiers using SimSiam's or DINO's representations obtained for the respective downstream datasets. To implement LSH, we rely on random projections [33] that we implement from scratch. For a detailed description of our stealing and downstream training parameters, we refer to Appendix F.

4.1 Local Sensitive Hashing for Coverage Estimation

We first observe that the choice of the total number of hash buckets in the LSH influences the effectiveness of our method. In the extreme, if we have a too large number of buckets, the number of buckets filled will correspond to the number of queries posed by a user which fails to capture that similar representations cover similar sub-spaces of the embedding space, and hence does not serve to approximate the total fraction of the embedding space covered. However, if we have too few buckets, even the representations for simple downstream tasks will fill large fractions of buckets, making it impossible to calibrate the cost function such that it only penalizes adversaries. We experimentally find that for our evaluated encoders, 2^{12} buckets represent a good trade-off. In Appendix F, we present an ablation study on the effect of the number of total buckets.

Our evaluation of the LSH to track coverage of the embedding space is presented in Figure 7a. We observe that representations returned for standard downstream tasks (FashionMNIST, SVHN, CIFAR10) occupy a significantly smaller fraction of the total number of buckets than complex data



(a) Fraction of Occupied Buckets.

(b) Number of Queries per Bucket.

Figure 3: **Estimating Embedding Space Coverage through LSH on SimSiam Encoder.** We present the fraction of buckets occupied by representations of different datasets as a function of the number of queries posed to the encoder *(left)*. We observe that representations for the downstream datasets (FashionMNIST, SVHN, CIFAR10) occupy a smaller fraction of buckets than representations from the complex ImageNet dataset. Our evaluation of the number of queries whose representations are mapped to the same bucket *(right)* indicates that our total number of buckets (2^{12}) is well calibrated for the estimation of covered representation space: over all datasets, we experience hash collisions, *i.e.,* queries whose representations are mapped to the same buckets. This indicates that our LSH is capable of representing similarities in the representations.

from multiple distributions (ImageNet). We further observe that the fraction of buckets occupied by the representations saturates over time. These findings highlight that LSH is successful in capturing the differences between legitimate users and adversaries—even in a low-query regime. Finally, we note that our total number of buckets (2^{12}) is well calibrated since, over all datasets, it successfully maps multiple representations to the same hash bucket while still filling various fractions of the total number of buckets.

4.2 Calibrating the Cost Function

We experiment with different sets of hyperparameters to instantiate the cost function from Equation (1) in the previous section (3.3). As described there, we can calibrate the function such that a desired penalty (in the form of a specific σ) will be assigned at a certain fraction of buckets occupied. For B4B, we aim at penalizing high embedding space coverage severely. Therefore, we need to identify and optimize for two components: 1) which value of σ leads to significant performance drops, and 2) for what fraction of coverage do we want to impose this significant drop. We base both components on empirical observations. Our first observation is that for our four downstream tasks (FashionMNIST, SVHN, STL10, and CIFAR10), performance drops to 10% (*i.e.,* random guessing) at roughly $\sigma = 0.5$. In Figure 7a, we further see that with 50k queries, the downstream tasks occupy $< 30\%$ of the buckets. Ultimately, setting α and β are design choices that an API provider needs to make in order to specify what type of query behavior they want to penalize. As very loose bounds (weak defense), based on our observation, we consider $\sigma = 1$ as a high penalty, which leads to $\alpha = 1$, and select $\beta = 0.8$. This β corresponds to considering 80% of buckets filled as a too-large coverage of the embedding space. We empirically observe that coverage of 80% of buckets occurs, for example, after around 100k of ImageNet queries. By choosing our target β so loose, *i.e.,* significantly larger than the observed 30% for downstream tasks, we offer flexibility for the API to also provide good representations for more complex downstream tasks. Finally, to obtain a flat cost curve close to the origin—which serves to map small fractions of covered buckets to small costs—we find that we can set $\lambda = 10^{-6}$. In the Appendix, we evaluate our defense end-to-end with differently parameterized cost functions.

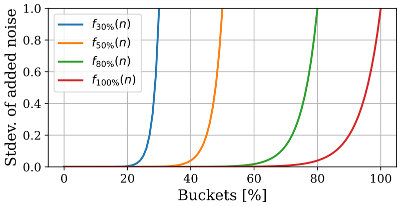


Figure 4: **Cost Function Calibration.**

4.3 Assessing the Effect of Transformations

Transformations Do Not Harm Utility for Legitimate Users. We evaluate the downstream accuracy for transformed representations based on training a linear classifier on top of them. To separate the effect of the noise added by our defense from the effect of the transformations, we perform the experiments in this subsection without adding noise to the returned representations. For example, on the CIFAR10 dataset and a SimSiam encoder pre-trained on ImageNet, without any transformations applied, we obtain a downstream accuracy of 90.41% (± 0.02), while, with transformations, we obtain 90.24% (± 0.11) for Affine, 90.40% (± 0.05) for Pad+Shuffle, 90.18% (± 0.06) for Affine+Pad+Shuffle, and 88.78% (± 0.2) for Binary. This highlights that the transformations preserve utility for legitimate users. This holds over all datasets we evaluate as we show in Appendix F.

Adversaries Cannot Perfectly Remap Representations over Multiple Sybil Accounts. To understand the impact of our per-user account transformations on sybil-attack based encoder stealing, we evaluate the difficulty of remapping representations between different sybil accounts. For simplicity, and since we established in Section 3.4 that multi-account attacks reduce to a two-account setup, we assume an adversary who queries from two sybil accounts and aims at learning to map the transformed representations from account #2 to the representation space of account #1. Using more accounts for the adversary causes a larger query overhead and potentially more performance loss from remapping. Our evaluation here, hence, represents a lower bound on the overhead caused to the adversary through our transformations.

We learn the mapping between different accounts' representations by training a linear model on overlapping representations between the accounts. We assess the fidelity of remapped representations as a function of the number of overlapping queries between the accounts. As a fidelity metric for our remapping, we compare the cosine distance between representations (a and b defined as: $1 - \frac{a^T b}{||a||_2 \cdot ||b||_2}$). Once the remapping is trained, we evaluate by querying 10k data points from the test dataset through account #1 and then again through account #2. Then, we apply the learned remapping to the latter one and compute the pairwise cosine distances between the representations from account #1 and their remapped counterparts from account #2. Our results are depicted in Figure 5. We show that the largest cosine distance is achieved with the binary transformations, making them the most protective against the adversary since they best prevent perfect remapping, even with an overlap of as many as 10k queries between both accounts. However, these binary transformations also incur the highest drop in accuracy for legitimate users. The defender has the possibility of selecting their preferred types of transformations between representations taking into account the trade-offs between the effectiveness of the defense and the negative impact on legitimate users.

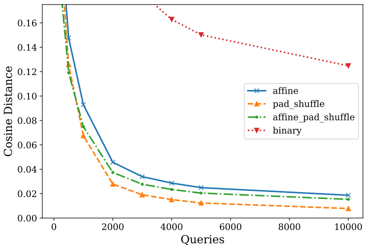


Figure 5: **Quality of Remappings.**

4.4 End-to-End Stealing of an Encoder under our Defense

We perform an end-to-end study to showcase how our B4B defense affects legitimate users vs adversaries. The hyperparameters for B4B are chosen according to the empirical evaluation of the previous sections with 2^{12} as the number of buckets, $\alpha = 1, \beta = 0.8, \lambda = 10^{-6}$ as the hyperparameter of the cost function, and different random affine transformations per-user account. Our main results are presented in Table 1. We observe that instantiating our framework with B4B has a negligible impact on legitimate users while substantially lowering the performance of stolen encoders in the case of single-user and sybil attackers.

Legitimate Users. We compare the accuracy of downstream classifiers trained on top of unprotected vs defended encoders. The victim encoder achieves high accuracy on the downstream tasks when no defense is employed. With B4B in place, we observe that across all the downstream tasks, the drop in performance is below 1%. For example, there is only a slight decrease in the accuracy of CIFAR10 from 90.41±0.02% to 90.24±0.11%. B4B's small effect on legitimate users stems from the fact that their downstream representations cover a relatively small part of the representations space. This results in a very low amount of noise added to their representations which preserves performance.

Table 1: **Stealing and Using Encoders With and Without our Defense.** The *USER* column represents the type of the APIs' user, where Legit denotes a legitimate user, Attack stands for a standard single-account adversary, and Sybil represents an adversary using two sybil accounts. We use InfoNCE loss for encoder extraction. # Queries stands for the number of queries used for stealing. The *TYPE* column expresses how the dataset is used. We follow the stealing setup from [16]. In the first row, we present the undefended victim encoder's performance as the accuracy for downstream tasks trained on the encoder's returned representations. In the following four rows, we show downstream utility for legitimate users when the victim encoder is defended by our B4B. Finally, (in the remaining rows) we assess the performance of stolen encoders on the downstream tasks. Our results highlight that while the performance of the encoder for legitimate users stays high, our B4B renders stealing inefficient with the stolen encoders obtaining significantly worse performance on downstream tasks.

User	Defense	# Queries	Dataset	Type	CIFAR10	STL10	SVHN	F-MNIST
N/A	N/A	*Victim*	*ImageNet*	*Train*	90.41±0.02	95.08±0.13	75.47±0.04	91.22±0.11
Legit	B4B	50k	CIFAR10	Query	90.24±0.11	N/A	N/A	N/A
Legit	B4B	5k	STL10	Query	N/A	95.05±0.1	N/A	N/A
Legit	B4B	73k	SVHN	Query	N/A	N/A	74.96±0.13	N/A
Legit	B4B	60k	F-MNIST	Query	N/A	N/A	N/A	91.7±0.01
Attack	None	50K	ImageNet	Steal	65.2 ±0.03	64.9 ±0.01	62.1 ±0.01	88.5 ±0.01
Attack	B4B	50k	ImageNet	Steal	35.72±0.04	31.54±0.02	19.74±0.02	70.01±0.01
Attack	None	100K	ImageNet	Steal	68.1 ±0.03	63.1 ±0.01	61.5 ±0.01	89.0 ±0.07
Attack	B4B	100k	ImageNet	Steal	12.01±0.07	13.94±0.05	19.96±0.03	69.63±0.07
Sybil	B4B	50k+50k	ImageNet	Steal	39.56± 0.06	38.50±0.04	23.41±0.02	77.01± 0.08

Adversaries. For adversaries who create a stolen copy of the victim encoder, we make two main observations. The most crucial one is that when our B4B is in place, the performance of the stolen copies over all downstream tasks significantly drops in comparison to when the victim encoder is unprotected (grey rows in Table 1). This highlights that our B4B effectively prevents stealing. Our next key observation concerns the number of stealing queries used by the adversary: When no defense is applied, the more queries are issued against the API (*e.g.,* 100k instead of 50k), the higher performance of the stolen encoder on downstream tasks (*e.g.,* CIFAR10 or FashionMNIST). In contrast, with B4B implemented as a defense, the performance decreases when using more stealing queries from a single account. This is because with more queries issued, the coverage of embedding space grows which renders the returned representations increasingly noisy and harms stealing performance. We also show that this performance drop cannot be prevented by sybil attacks. Therefore, we consider an adversary who queries from two sybil accounts with 50k queries issued per account and the first 10k queries of both accounts used to learn the remapping of representations between them. When the adversary trains their stolen encoder copy on all the remapped representations, they increase downstream performance over querying from a single account. Yet, their performance is still significantly smaller than the performance of the victim encoder for legitimate users, or the encoder stolen from an undefended victim. This highlights that our B4B also successfully prevents sybil attacks.

5 Conclusions

We design B4B a new and modular active defense framework against stealing SSL encoders. All the previous approaches were either reactive, acting after the attack happened to detect stolen encoders, or lowered the quality of outputs substantially also for legitimate users which rendered such mechanisms impractical. We show that B4B successfully distinguishes between legitimate users and adversaries by tracking the embedding space coverage of users' obtained representations. B4B then leverages this tracking to apply a cost function that penalizes users based on the current space coverage, for instance, by lowering the quality of their outputs. Finally, B4B prevents sybil attacks by implementing per-user transformations for the returned representations. Through our experimental evaluation, we show that our defense indeed renders encoder stealing inefficient while preserving downstream utility for legitimate users. Our B4B is therefore a valuable contribution to a safer sharing and democratization of high-utility encoders over public APIs.

References

[1] Clarifai, https://www.clarifai.com/. URL `https://www.clarifai.com/`.

[2] Cohere, https://cohere.ai. URL `https://cohere.ai/`.

[3] Openai, https://openai.com. URL `https://openai.com/`.

[4] Poet2, https://labs.hyperledger.org/labs/archived/sawtooth-poet2.html. URL `https://labs.hyperledger.org/labs/archived/sawtooth-poet2.html`.

[5] Google, https://google.github.io/snappy/. URL `https://google.github.io/snappy/`.

[6] Facebook zstd, https://github.com/facebook/zstd. URL `https://github.com/facebook/zstd`.

[7] Yossi Adi, Carsten Baum, Moustapha Cisse, Benny Pinkas, and Joseph Keshet. Turning your weakness into a strength: Watermarking deep neural networks by backdooring. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1615–1631, 2018.

[8] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom B Brown, Dawn Song, Ulfar Erlingsson, et al. Extracting training data from large language models. In *USENIX Security Symposium*, volume 6, 2021.

[9] Mathilde Caron, Hugo Touvron, Ishan Misra, Hervé Jégou, Julien Mairal, Piotr Bojanowski, and Armand Joulin. Emerging properties in self-supervised vision transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 9650–9660, October 2021.

[10] Ting Chen, S. Kornblith, M. Norouzi, and G. Hinton. A simple framework for contrastive learning of visual representations. *International Conference on Machine Learning*, 2020.

[11] Xinlei Chen and Kaiming He. Exploring simple siamese representation learning. 2020.

[12] Adam Coates, Andrew Ng, and Honglak Lee. An Analysis of Single Layer Networks in Unsupervised Feature Learning. In *AISTATS*, 2011. `https://cs.stanford.edu/~acoates/papers/coatesleeng_aistats_2011.pdf`.

[13] Tianshuo Cong, Xinlei He, and Yang Zhang. Sslguard: A watermarking scheme for self-supervised learning pre-trained encoders. *CoRR*, abs/2201.11692, 2022. URL `https://arxiv.org/abs/2201.11692`.

[14] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[15] Adam Dziedzic, Nikita Dhawan, Muhammad Ahmad Kaleem, Jonas Guan, and Nicolas Papernot. On the difficulty of defending self-supervised learning against model extraction. In *International Conference on Machine Learning*, 2022.

[16] Adam Dziedzic, Haonan Duan, Muhammad Ahmad Kaleem, Nikita Dhawan, Jonas Guan, Yannis Cattan, Franziska Boenisch, and Nicolas Papernot. Dataset inference for self-supervised models. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL `https://openreview.net/forum?id=CCBJf9xJo2X`.

[17] Adam Dziedzic, Muhammad Ahmad Kaleem, Yu Shen Lu, and Nicolas Papernot. Increasing the cost of model extraction with calibrated proof of work. In *International Conference on Learning Representations*, 2022. URL `https://arxiv.org/abs/2201.09243`.

[18] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. 2013. URL `https://eprint.iacr.org/2013/796`. `https://eprint.iacr.org/2013/796`.

[19] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. In *Advances in Cryptology–CRYPTO 2015: 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, pages 585–605. Springer, 2015.

[20] Nicholas Frosst, Nicolas Papernot, and Geoffrey Hinton. Analyzing and improving representations with the soft nearest neighbor loss. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2012–2020. PMLR, 09–15 Jun 2019. URL `https://proceedings.mlr.press/v97/frosst19a.html`.

[21] Jean-Bastien Grill, F. Strub, F. Altché, C. Tallec, P. H. Richemond, E. Buchatskaya, C. Doersch, B. A. Pires, Z. D. Guo, M. G. Azar, B. Piot, K. Kavukcuoglu, R. Munos, and M. Valko. Bootstrap your own latent: A new approach to self-supervised learning. *Computer Vision and Pattern Recognition*, 2020.

[22] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 16000–16009, June 2022.

[23] Matthew Jagielski, N. Carlini, D. Berthelot, A. Kurakin, and N. Papernot. High accuracy and high fidelity extraction of neural networks. *USENIX Security Symposium*, 2020.

[24] Hengrui Jia, Christopher A Choquette-Choo, Varun Chandrasekaran, and Nicolas Papernot. Entangled watermarks as a defense against model extraction. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1937–1954, 2021.

[25] Mika Juuti, Sebastian Szyller, Samuel Marchal, and N Asokan. Prada: protecting against dnn model stealing attacks. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 512–527. IEEE, 2019.

[26] Sanjay Kariyappa, Atul Prakash, and Moinuddin K Qureshi. Maze: Data-free model stealing attack using zeroth-order gradient estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13814–13823, 2021.

[27] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[28] Yupei Liu, Jinyuan Jia, Hongbin Liu, and Neil Zhenqiang Gong. Stolenencoder: Stealing pre-trained encoders in self-supervised learning. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 2115–2128, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394505. doi: 10.1145/3548606.3560586. URL `https://doi.org/10.1145/3548606.3560586`.

[29] Pratyush Maini, Mohammad Yaghini, and Nicolas Papernot. Dataset inference: Ownership resolution in machine learning. In *Proceedings of ICLR 2021: 9th International Conference on Learning Representationsn*, 2021.

[30] Yuki Nagai, Y. Uchida, S. Sakazawa, and Shin'ichi Satoh. Digital watermarking for deep neural networks. *International Journal of Multimedia Information Retrieval, 7:3–16*, 2018.

[31] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*, 2011. URL `http://ufldl.stanford.edu/housenumbers/nips2011_housenumbers.pdf`.

[32] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. Knockoff nets: Stealing functionality of black-box models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4954–4963, 2019.

[33] Loïc Paulevé, Hervé Jégou, and Laurent Amsaleg. Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern recognition letters*, 31(11):1348–1358, 2010.

[34] Zeyang Sha, Xinlei He, Ning Yu, Michael Backes, and Yang Zhang. Can't steal? cont-steal! contrastive stealing attacks against image encoders. 2022. URL `https://arxiv.org/abs/2201.07513`.

[35] Yun Shen, Xinlei He, Yufei Han, and Yang Zhang. Model stealing attacks against inductive graph neural networks. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1175–1192. IEEE, 2022.

[36] Ram Shankar Siva Kumar, Magnus Nyström, John Lambert, Andrew Marshall, Mario Goertzel, Andi Comissoneru, Matt Swann, and Sharon Xia. Adversarial machine learning-industry perspectives. In *2020 IEEE Security and Privacy Workshops (SPW)*, pages 69–75, 2020. doi: 10.1109/SPW50608.2020.00028.

[37] Malcolm Slaney and Michael Casey. Locality-sensitive hashing for finding nearest neighbors [lecture notes]. *IEEE Signal processing magazine*, 25(2):128–131, 2008.

[38] Florian Tramèr, F. Zhang, A. Juels, M. Reiter, and T. Ristenpart. Stealing machine learning models via prediction apis. *USENIX Security Symposium*, 2016.

[39] Jean-Baptiste Truong, Pratyush Maini, Robert J. Walls, and Nicolas Papernot. Data-free model extraction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2021.

[40] Yusuke Uchida, Yuki Nagai, Shigeyuki Sakazawa, and Shin'ichi Satoh. Embedding watermarks into deep neural networks. In *Proceedings of the 2017 ACM on international conference on multimedia retrieval*, pages 269–277, 2017.

[41] Yutong Wu, Han Qiu, Tianwei Zhang, Lin Jiwei, and Meikang Qiu. Watermarking pre-trained encoders in contrastive learning. *ArXiv*, abs/2201.08217, 2022.

[42] Jure Zbontar, Li Jing, Ishan Misra, Yann LeCun, and Stéphane Deny. Barlow twins: Self-supervised learning via redundancy reduction. *arXiv preprint arXiv:2103.03230*, 2021.

A Broader Impacts

The goal of our work is to actively defend self-supervised encoders against model stealing attacks. Since we are directly defending encoders, any negative societal impacts of our work are minimal. One potentially negative impact could be the degradation of performance for legitimate users. However, as shown in our experimental results, we are able to preserve high utility for standard users.

B Limitations

We show how our defense method is tuned for SimSiam and DINO. There are more types of SSL encoders that can be tested with our method. The B4B defense method requires tuning the parameters, such as the number of occupied buckets that is allowed without any penalty for the cost function, or the selection of the transformations. These steps are rather difficult to automate but can be replaced with more data-driven approaches. For example, instead of designing a cost function from scratch, we could create an ML model to obtain a cost for a given occupation of the representation space. We explain more details in the Appendix C.2.

C Alternative Building Blocks to Instantiate B4B

While we present a reference implementation of B4B in our work that instantiates the three building blocks with (1) Local Sensitive Hashing, (2) Utility of the Representations, and (3) a set of concrete transformations, there exists a multitude of alternatives to concretely implement our B4B framework. In the following, we present these alternatives, grouped by building block.

C.1 Alternative Estimation of the Coverage of Embedding Space

We also explore alternative methods to measure the distances between representations for queries sent to an API. One of them is to apply the cosine distance (where for two representations a and b, it is defined as: $1 - \frac{a^T b}{||a||_2 \cdot ||b||_2}$) since it can be measured between individual data points in a pair-wise fashion. If the total pair-wise cosine distance between representations for a given user is small, then the user queries presumably come from a single downstream task distribution. Otherwise, a user might be malicious and would like to cover a large part of the representation space, then the total pair-wise cosine distance for the user's representations would be high. Note that in this case, the cosine distance can be replaced with any other distance measure, such as the Minkowski distance. We opt for the LSH in our reference implementation, since it is much less expensive to compute than cosine distance. LSH requires only $2^{12} = 4096$ buckets that can be expressed as a binary table with the same number of elements, which requires in the worst case iterating over all of them to count how many are occupied. With more than 4096 queries sent by a given user, the computation on the LSH is sublinear $< O(n)$ with respect to the number of user queries. For the cosine distance approach, the required computation grows quadratically $O(n^2)$ with the number of queries.

C.2 Alternative Cost Functions

The cost functions can be designed from scratch manually or learned, for example, via an ML model, such as a neural network or SVM. In our initial version, the function was designed manually, where the underlying premise is that once a specified number of buckets is occupied, the cost should grow exponentially. Instead of defining such a function or providing the high-level parameters for functions that we contributed, one could learn an ML model that for a given number of buckets occupied, it should output an estimated cost, or even directly, the desired σ (standard deviation) of the noise added to representations. This method requires a relatively large number of data points to be provided for training the model, however, lowers the burden on a defender to either decide on the specific function or adjust its parameters. Thus, it could be more user-friendly, for example, not necessitating any mathematical background, but can be precise enough to obtain the desired behavior.

Note that instead of adding the calibrated noise (proportional to the estimated cost) to the representations, we could rather require a given user to pay a higher monetary cost for queries that cover a large fraction of the representation space, or force a user to solve a puzzle in a form of the proof-of-work [17], wait a specified amount of time via proof-of-elapsed time (PoET) [4], or prove that a specified amount of disk space was reserved [18, 19]. For example, consider the approach with PoET. A user sends queries to the API, which we cost based on their occupation of the embedding space. The user is sent a waiting time. The users' resource (e.g., a CPU) has to be occupied for this specific waiting time without performing any work. At the end of the specified amount of time, the user sends proof of the elapsed time, which can be easily verified by the server. PoET requires access to specialized hardware, for example, secure CPU instructions that are becoming widely available in consumer and enterprise processors. If a user does not own such hardware, proof of elapsed time could be produced using a service exposed by a cloud provider (e.g., Azure VMs that feature TEE 2). Note that if a server sends the time based on the calculated cost, the adversary might learn the cost function. Instead, the exact waiting time should be split in random *subwaiting* times and sent to the user one by one. Thus, a server should rather have a few rounds of exchange with the client to incur the additional cost.

C.3 Alternative to Transformations

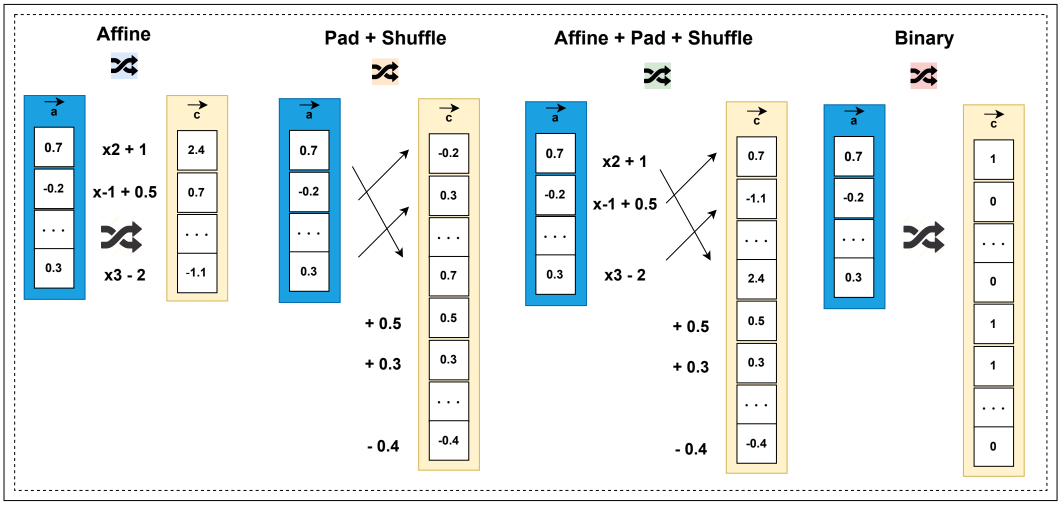


Figure 6: **Overview on Transformations.** We depict the inner-workings of the transformations considered in this work.

As an alternative to the transformations used within this work (see Figure 6), one could use a different set of transformations or combinations thereof. The padding can be done with different constant values and combined with adding constant values within the representations. The padding and adding the constant values can be followed by shuffling the elements within the representations. We can apply the affine of binary transformations on top of the padding and shuffling. Additionally, we can also use other pre-defined linear transformations like rotations or shearing.

The representations could also be compressed to smaller vectors and the compression rate would depend on the occupation of the representation space, for example, the higher the number of occupied buckets in our hash table, the more compressed the output representations could be. Such representations could be compressed via FFT, a cosine transform, or standard compression techniques such as snappy [5]. If the information from the representations should not be lost, then the lossless compression techniques can be applied, for instance, zstd [6]. The only requirement of the compression techniques is to ensure that they do not decrease the accuracy on downstream tasks for legitimate users.

Another alternative is to incorporate an additional neural network layer for transforming the returned representations. The training of this supplementary layer should primarily focus on preserving the usability of the representations for legitimate users. This approach grants the API provider with additional capabilities, as it allows for the utilization of customized training objectives. For instance, if the API provider employs LSH (Locality-Sensitive Hashing) to estimate the coverage of the representation space, they can leverage buckets and train the additional layer to maintain high-quality representations exclusively for frequently-used buckets and their surrounding areas, while not prioritizing the rest of the representation space. This approach safeguards legitimate users from any adverse effects, as their coverage of the representation space is minimal. Simultaneously, it ensures that adversaries are unable to exploit representations from the entire representation space.

D Sybil Attacks

We consider an adversary who generates n sybil accounts to steal the encoder from the API. For each of the accounts, the representations are transformed in a different way. Therefore, to replicate the victim model using all the obtained representations, the adversary has to map these representations into one single space. This can be done, for example, by training a neural network to perform the mapping.

We assume the adversary obtains $\{N_1, N_2, \ldots, N_n\}$ many representations from the victim for each of the n sybil accounts. Without loss of generality, we assume the adversary maps them back to the embedding space of the first sybil account. To learn the mapping, the adversary can apply different strategies.

D.1 Sybil Strategies

We present three potential approaches that Sybils might want to apply to circumvent our defense. Consider three users: A, B, and C, with their respective datasets D_A, D_B, and D_C, each with different distributions to maximize extraction effectiveness. First, user A is selected to unify representations from other users B and C. User A would have to query from at least two different datasets D_B and D_C, while other users would act legitimately. Sybil attackers want to deploy as many users as possible but with more fake accounts, user A incurs high coverage of the representation space, and this is prevented by our single-user defense. In all other cases neither of the sybil users can act legitimately, thus they are already affected by the single-user defense. Second, user A would query from their own dataset D_A and partially from dataset D_B. Then user B would query from their own dataset D_B and partially from dataset D_C, and so on. This method is the most inconspicuous but requires a number of remappings that scales super-exponentially with the number of fake accounts, which is impractical. Finally, each user would query from their respective dataset, for example, user B would query from dataset D_B and additionally from a remapping dataset, *e.g.,* D_A. Representations could be unified by mapping them to A's representations. The last approach as well as all other cases reduce to the minimum of remapping between representations of a pair of users. We show that our defense cuts such attempts short by ensuring that the remapping between representations is prohibitive even for a pair of users.

E Additional Related Work

One of the main workhorse techniques used in the encoders is contrastive learning, where the representations are trained so that the positive pairs (two augmented versions of the same image) have similar representations while negative pairs (augmentations of two different images) have representations which are far apart.

SimSiam utilizes Siamese networks (two encoders with shared weights) but with a simplified training process and architecture. In contrast to the previous frameworks, such as SimCLR [10], SimSiam's authors show that negative samples are unnecessary and collapsing solutions can be avoided by applying the projection head to of one of the encoders, and a stop-gradient operation to the other. SimSiam minimizes the negative cosine similarity between two randomly augmented views of the same image from the Siamese encoders, which is expressed via a symmetrized loss [21]. This creates a simple yet highly effective representation learning method.

DINO is another popular representation learning framework. While SimSiam uses CNNs, DINO employs vision transformers (ViTs). It trains a student and teacher encoder with the same architecture, updating the teacher with an (exponential moving) average of the student. Different random transformations of the same image are passed through both encoders. The student receives smaller image crops, forcing it to generate representations restoring parts of the original image. The training objective is minimizing cross-entropy loss between teacher and student representations.

F Additional Experimental Results

F.1 Details on Experimental Setup

The end-to-end experiments on stealing SimSiam and ViT DINO encoders were done using 3 A100 GPUs. Detailed experiments including mapping, transformations and the evaluation was performed using a single computer equipped with two Nvidia RTX 2080 Ti GPUs.

F.2 Datasets Used

CIFAR10 [27]: The CIFAR10 dataset consists of 32x32 colored images with 10 classes. There are 50000 training images and 10000 test images.

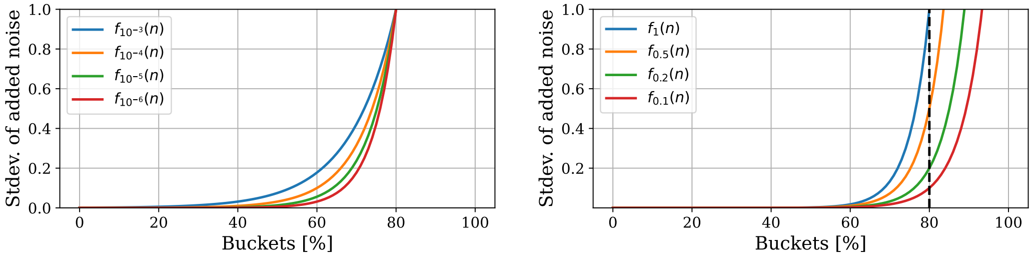
SVHN [31]: The SVHN dataset contains 32x32 coloured images with 10 classes. There are roughly 73000 training images, 26000 test images and 530000 "extra" images.

ImageNet[14]: Larger sized coloured images with 1000 classes. As is commonly done, we resize all images to be of size 224x224. There are approximately 1 million training images and 50000 test images.

STL10 [12]: The STL10 dataset contains 96x96 coloured images with 10 classes. There are 5000 training images, 8000 test images, and 100000 unlabeled images.

F.3 More Results for the End2End Empirical Evaluation

We consider fine-tuning parameters β, λ, and α for our cost function and the intuitive meaning behind these parameters. In general, our recommendation is to adjust the parameter β that specifies how many buckets are allowed to be filled by users' downstream tasks. On the other hand, when parameter λ is increased, this causes a higher amount of added noise before we reach the number of buckets specified by β, which lowers the performance of a given downstream task relatively early. For example, a higher value of λ in Figure 4, would cause an increase in the amount of added noise much earlier than for the target value of β. Finally, parameter α controls the amount of noise once the number of buckets specified by β is reached. Thus, in Figure 4, we set $\alpha = 1$ and the maximum standard deviation of the added Gaussian noise is 1.



(a) Cost Function for different λ parameter values.

(b) Cost Function for different α parameter values.

Figure 7: **Effects of λ and α parameters on the Cost Function.** We present the Cost Function for α=1, β=80 and different values of λ (*left*) and $\lambda = 10^{-6}$, β=80 and different values of α (*right*).

Table 2: **Stealing and Using Encoders With and Without our Defense**. The model used in the experiments is Simsiam, with the following parameters for the cost function $\lambda = 10^{-4}$, $\alpha = 1$, and $\beta = 80\%$, and the number of buckets equal to 2^{12}. Due to the higher value of the parameter λ, we observe lower performance on downstream tasks for the attackers since the magnitude of noise added to the representations is higher. However, for more complicated tasks than CIFAR10, this change might cause a potential drop in accuracy for the legitimate users.

User	Defense	# Queries	Dataset	Type	CIFAR10	STL10	SVHN	F-MNIST
N/A	N/A	*Victim*	*ImageNet*	*Train*	90.41±0.02	95.08±0.13	75.47±0.04	91.22±0.11
Legit	B4B	50k	CIFAR10	Query	90.02 ±0.1	N/A	N/A	N/A
Legit	B4B	5k	STL10	Query	N/A	94.88 ±0.17	N/A	N/A
Legit	B4B	73k	SVHN	Query	N/A	N/A	74.72 ±0.13	N/A
Legit	B4B	60k	F-MNIST	Query	N/A	N/A	N/A	91.76 ±0.09
Attack	None	50K	ImageNet	Steal	65.2 ±0.03	64.9 ±0.01	62.1 ±0.01	88.5 ±0.01
Attack	B4B	50k	ImageNet	Steal	28.22 ± 0.04	26.62 ± 0.02	19.62 ± 0.02	78.41 ±0.01
Attack	None	100K	ImageNet	Steal	68.1 ±0.03	63.1 ±0.01	61.5 ±0.01	89.0 ±0.07
Attack	B4B	100k	ImageNet	Steal	17.73 ± 0.18	15.59 ± 0.61	19.53± 0.01	55.11 ± 0.05
Sybil	B4B	50k+50k	ImageNet	Steal	33.43 ± 0.03	31.18 ± 0.12	22.91 ± 0.01	75.35 ± 0.05

Table 3: **Stealing and Using Encoders With and Without our Defense**. The model used in the experiments is Simsiam, with the following parameters for the cost function $\lambda = 10^{-6}$, $\alpha = 1$, and $\beta = 50\%$, and the number of buckets equal to 2^{12}. This experiment corresponds to considering 50% of buckets filled as a too-large coverage of the embedding space. This improves the defense but again might potentially harm the performance of more complicated tasks than CIFAR10 since they could occupy more buckets than 50%.

User	Defense	# Queries	Dataset	Type	CIFAR10	STL10	SVHN	F-MNIST
N/A	N/A	*Victim*	*ImageNet*	*Train*	90.41±0.02	95.08±0.13	75.47±0.04	91.22 ±0.11
Legit	B4B	50k	CIFAR10	Query	90.27 ±0.07	N/A	N/A	N/A
Legit	B4B	5k	STL10	Query	N/A	95.12 ±0.13	N/A	N/A
Legit	B4B	73k	SVHN	Query	N/A	N/A	74.94 ±0.16	N/A
Legit	B4B	60k	F-MNIST	Query	N/A	N/A	N/A	91.66 ±0.05
Attack	None	50K	ImageNet	Steal	65.2 ±0.03	64.9 ±0.01	62.1 ±0.01	88.5 ±0.01
Attack	B4B	50k	ImageNet	Steal	15.52 ± 0.37	12.57 ± 0.23	19.53± 0.01	23.17 ± 0.01
Attack	None	100K	ImageNet	Steal	68.1 ±0.03	63.1 ±0.01	61.5 ±0.01	89.0 ±0.07
Attack	B4B	100k	ImageNet	Steal	16.27 ± 0.04	13.93 ± 0.35	19.54 ± 0.02	54.69 ± 0.02
Sybil	B4B	50k+50k	ImageNet	Steal	30.14 ± 0.01	29.57 ± 0.08	19.99 ± 0.03	71.72 ± 0.01

Table 4: **Stealing and Using Encoders With and Without our Defense**. The model used in the experiments is Simsiam, with the following parameters for the cost function $\lambda = 10^{-6}$, $\alpha = 1$, and $\beta = 30\%$, and the number of buckets equal to 2^{12}. Since the value of parameter β is decreased substantially to 30%, we observe a drop in accuracy for legitimate users. For example, more than 1% for CIFAR10. In the next Table 5, we show that by also decreasing the parameter α, we can attenuate this harmful effect and retain higher accuracy for legitimate users. In case of an attack, for 100k stealing queries, we observe much lower accuracy levels than for $\beta = 50\%$ shown in Table 3.

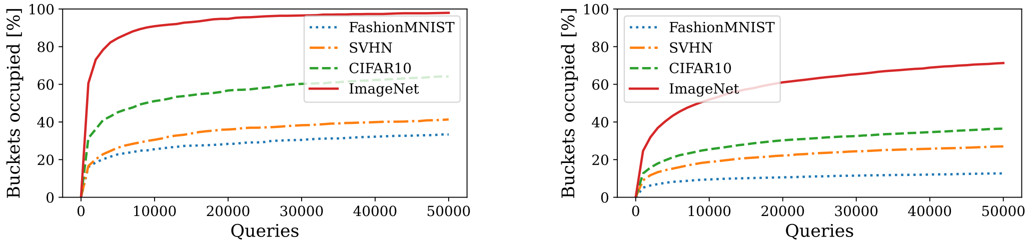
User	Defense	# Queries	Dataset	Type	CIFAR10	STL10	SVHN	F-MNIST
N/A	N/A	*Victim*	*ImageNet*	*Train*	90.41±0.02	95.08±0.13	75.47±0.04	91.22 ±0.11
Legit	B4B	50k	CIFAR10	Query	88.1 ±0.11	N/A	N/A	N/A
Legit	B4B	5k	STL10	Query	N/A	94.92 ±0.11	N/A	N/A
Legit	B4B	73k	SVHN	Query	N/A	N/A	74.37 ±0.02	N/A
Legit	B4B	60k	F-MNIST	Query	N/A	N/A	N/A	91.67 ±0.07
Attack	None	50K	ImageNet	Steal	65.2 ±0.03	64.9 ±0.01	62.1 ±0.01	88.5 ±0.01
Attack	B4B	50k	ImageNet	Steal	30.82 ± 0.09	26.37 ± 0.07	21.87 ± 0.03	66.0 ± 0.02
Attack	None	100K	ImageNet	Steal	68.1 ±0.03	63.1 ±0.01	61.5 ±0.01	89.0 ±0.07
Attack	B4B	100k	ImageNet	Steal	9.57 ± 0.17	9.83 ± 0.09	19.57 ± 0.01	27.06 ± 0.46
Sybil	B4B	50k+50k	ImageNet	Steal	29.15 ± 0.02	28.67 ± 0.06	19.98 ± 0.03	70.62 ± 0.03

Table 5: **Stealing and Using Encoders With and Without our Defense**. The model used in the experiments is Simsiam, with the following parameters for the cost function $\lambda = 10^{-6}$, $\alpha = 0.1$, and $\beta = 30\%$, and the number of buckets equal to 2^{12}. Due to the lower performance on downstream tasks observed in Table 4 while keeping the parameter β fixed to 30% and λ fixed to 10^{-6}, we decrease the value of parameter α to 0.1, which increases the performance of legitimate users on their downstream tasks. In this experiment, we also carry out a sybil attack with more accounts (4 instead of 2), but observe that this modification does not improve the performance of the attacker. With more accounts, a sybil has to sacrifice more queries for the remappings between the representations from different accounts. Additionally, note that each account introduces a different remapping error by the dint of different transformations applied to each account by B4B.

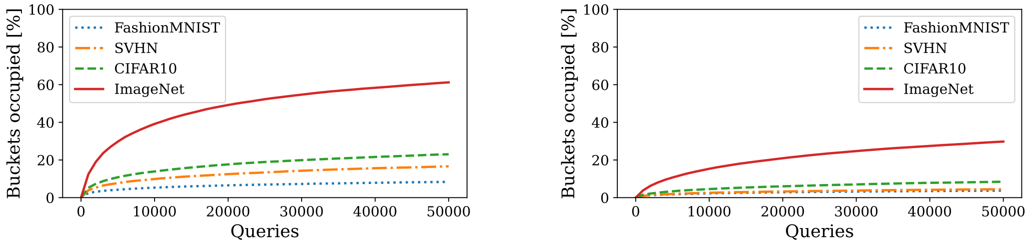
User	Defense	# Queries	Dataset	Type	CIFAR10	STL10	SVHN	F-MNIST
N/A	N/A	*Victim*	*ImageNet*	*Train*	90.41±0.02	95.08±0.13	75.47±0.04	91.22±0.11
Legit	B4B	50k	CIFAR10	Query	90.17 ±0.1	N/A	N/A	N/A
Legit	B4B	5k	STL10	Query	N/A	94.92 ±0.09	N/A	N/A
Legit	B4B	73k	SVHN	Query	N/A	N/A	74.97 ±0.13	N/A
Legit	B4B	60k	F-MNIST	Query	N/A	N/A	N/A	91.71 ±0.08
Attack	None	50K	ImageNet	Steal	65.2 ±0.03	64.9 ±0.01	62.1 ±0.01	88.5 ±0.01
Attack	B4B	50k	ImageNet	Steal	19.95 ±0.19	15.54 ± 0.34	19.57 ± 0.01	23.50 ± 0.19
Attack	None	100K	ImageNet	Steal	68.1 ±0.03	63.1 ±0.01	61.5 ±0.01	89.0 ±0.07
Attack	B4B	100k	ImageNet	Steal	10.35 ± 0.19	12.37 ± 0.69	19.34 ± 0.01	68.93 ± 0.17
Sybil	B4B	4×25k	ImageNet	Steal	33.15 ± 0.04	30.23 ± 0.07	20.87 ± 0.01	72.19 ± 0.02

F.4 Setting the number of buckets

We present our procedure to find an optimal number of buckets in Figure 8.



(a) **Number of buckets =** 2^8 (b) **Number of buckets =** 2^{10}



(c) **Number of buckets =** 2^{12} (d) **Number of buckets =** 2^{14}

Figure 8: **Estimating Embedding Space Coverage through LSH on SimSiam Encoder.** We extend the results from Figure 7a(a) and present the fraction of buckets occupied by representations of different datasets as a function of the number of queries posed to the encoder. We consider different number of buckets in the LSH table. We observe that 2^8 buckets is to small since queries from the ImageNet dataset saturate all the buckets after around 50k queries, while the number 2^{14} of buckets is too large since it is never occupied more than 40%. Thus, the number 2^{12} buckets is a good middle ground. Subfigure (c) corresponds to Figure 7a from the main paper. We also use the same notation and carry out our experiments in the same way as in Figure 7a.

F.5 Results DINO

We show that our defense is also applicable to the DINO encoder. The occupation of the representations space is presented visually in Figure 9. We also show that the number of buckets 2^{12} is optimal for DINO in Figure 10. The impact of transformation on the representations from DINO is shown Table 7. Finally, the end to end experiment for DINO is presented in Table 6.

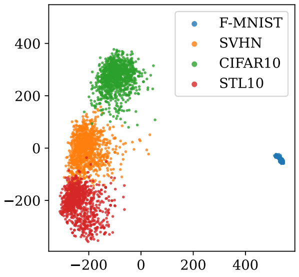


Figure 9: **Representations from Different Tasks Occupy Different Sub-Spaces of the Embedding Space. Presented for FashionMNIST, SVHN, CIFAR10, and STL10.** In this plot, we used the DINO ViT Small encoder trained on ImageNet.

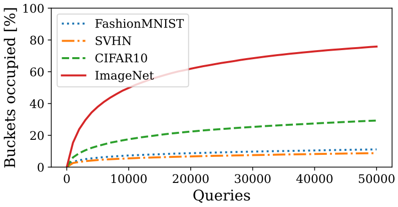


Figure 10: **Estimating Embedding Space Coverage through LSH on the DINO Encoder.** The number of buckets is set to 2^{12}. We also use the same notation and carry out our experiments in the same way as in Figure 7a.

Table 6: **Stealing and Using Encoders With and Without our Defense**. The model used in the experiments is DINO, with the following parameters for the cost function $\lambda = 10^{-6}$, $\alpha = 1000$, and $\beta = 60\%$, and the number of buckets equal to 2^{12}. We have to increase the value of parameter α by ×1000 since the norms of the DINO representations are also around 10^3 higher than for SimSiam. We observe that B4B performs similarly on DINO as for SimSiam.

User	Defense	# Queries	Dataset	Type	CIFAR10	STL10	SVHN	F-MNIST
N/A	N/A	*Victim*	*ImageNet*	*Train*	94.51 ±0.08	97.98 ±0.04	70.66 ±0.16	89.98 ±0.03
Legit	B4B	50k	CIFAR10	Query	94.25 ±0.11	N/A	N/A	N/A
Legit	B4B	5k	STL10	Query	N/A	98.05 ±0.04	N/A	N/A
Legit	B4B	73k	SVHN	Query	N/A	N/A	69.66 ±0.14	N/A
Legit	B4B	60k	F-MNIST	Query	N/A	N/A	N/A	89.68 ±0.01
Attack	None	50K	ImageNet	Steal	67.92 ± 0.04	66.02 ± 0.22	61.30 ± 0.01	89.46 ± 0.01
Attack	B4B	50k	ImageNet	Steal	42.02±0.05	38.91±0.06	19.94±0.02	73.33±0.04
Attack	None	100K	ImageNet	Steal	75.07 ± 0.01	76.32 ± 0.02	71.79 ± 0.06	89.76 ± 0.01
Attack	B4B	100k	ImageNet	Steal	19.27±0.03	21.24±0.03	19.84±0.01	71.01±0.03
Sybil	B4B	50k+50k	ImageNet	Steal	45.56± 0.06	42.50±0.02	24.25±0.03	78.01± 0.08

F.6 Additional evaluation of transformations

Additionally, we show the impact of transformations on the performance of legitimate users in Table 7 (for both SimSiam and DINO).

Table 7: **Impact of Transformations on the Performance for Legitimate Users.** We show that the transformations applied per-account do not harm the performance of legitimate users on their downstream tasks. The victim encoders was trained on the ImageNet dataset using SimSiam and DINO frameworks.

Transformation	Encoder	CIFAR10	STL10	SVHN	F-MNIST
None	*Victim SimSiam*	90.41±0.02	95.08 ±0.13	75.47 ±0.04	91.22 ±0.11
Affine	SimSiam	90.24 ±0.11	95.05 ±0.1	74.96 ±0.18	91.42 ±0.15
Pad+Shuffle	SimSiam	90.4 ±0.05	95.34 ±0.06	75.47 ±0.01	91.38 ±0.15
Affine+Pad+Shuffle	SimSiam	90.18 ±0.06	95.03 ±0.05	74.86 ±0.1	91.35 ±0.1
Binary	SimSiam	88.78 ±0.2	94.72 ±0.02	68.42 ±0.16	88.91 ±0.34
None	*Victim DINO*	94.51 ±0.08	97.98 ±0.04	70.66 ±0.16	89.98 ±0.03
Affine	DINO	94.25 ±0.11	98.05 ±0.04	69.77 ±0.11	89.68 ±0.01
Pad+Shuffle	DINO	94.72 ±0.02	98.07 ±0.03	70.44 ±0.1	89.91 ±0.08
Affine+Pad+Shuffle	DINO	94.26 ±0.06	98.02 ±0.01	69.49 ±0.2	89.70 ±0.1
Binary	DINO	92.96 ±0.1	98.03 ±0.03	59.53 ±0.27	88.26 ±0.04

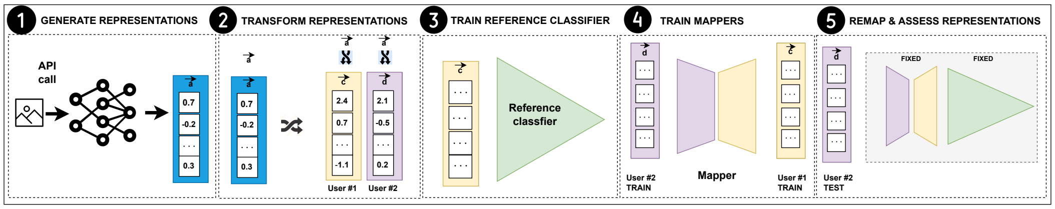


Figure 11: **Protocol to Evaluate the Mapping Between Representations.** We present the protocol of evaluating remappings for two sybil accounts. ❶ API receives inputs from two sybil accounts and generates corresponding representations. ❷ Representations are transformed on a per-user basis and returned. ❸ Adversary trains a reference classifier on representations from account one. ❹ Adversary trains a linear model to find mapping from representations of account two to representations of account one. ❺ To check the quality of obtained mapping representations from test set of account two are mapped using the fixed mapper (from step 4) to representation space of account one. This enables the calculation of cosine distance between representations from account one and their counterparts from account two shown in Figure 5. Additionally, the fixed reference classifier (from step 3) can be used to measure the accuracy drop caused by remapping .