SURPRISE MASKING: PRIVATE DATA FILTERING WITH MODEL LIKELIHOODS

Anonymous authors
Paper under double-blind review

ABSTRACT

Data useful for language model (LM) training can contain private or unwanted information that is at risk of being learned or memorized. We propose Surprise Masking (SM), a simple form of perplexity filtering that uses a publicly available reference model to identify unwanted spans and suppress learning from them while preserving the rest of each example. On synthetic code benchmarks, we show SM improves the privacy-utility tradeoff over standard masking methods and PII filtering. In fine-tuning experiments, SM reduces the recovery of private values from 66.3% to 2.3% and the reproduction of stylistic coding quirks from 23.6% to 1.3%, while preserving performance on the evaluated coding tasks. SM is similarly useful in pretraining. We also evaluate SM for sanitizing datasets by directly removing unwanted content. Here, SM outperforms DePA, a perplexity-based defense for detecting poisoned dead code, while specialized detectors perform better on broader credential and PII benchmarks. Across eleven settings, we find that SM's downstream success is predictable from the separation of unwanted and useful content in the distribution; particularly, we find it particularly useful with code data. Ultimately, SM is an inexpensive, adaptable method to preserve valuable examples while reducing the learning of unwanted information.

1 INTRODUCTION

Training samples that are valuable for learning can also contain undesirable content, including private information. This leaves practitioners with a few options: discarding these samples reduces leakage risk but sacrifices useful information, while training on them requires safeguards with their own costs and limitations. Existing safeguards include data-side interventions, which can be error-prone, expensive, or limited to narrow distributions (Qi et al., 2021; Tsai et al., 2025) Alternatively, training-time interventions alter how models learn from risky data (Hans et al., 2024; Wichers et al., 2025; Tan et al., 2025) without explicitly identifying the affected content, making their privacy protections harder to trust. Post-training unlearning methods typically localize information within a model before removing it (Merullo et al., 2025; Maini et al., 2023; Stoehr et al., 2024), which can

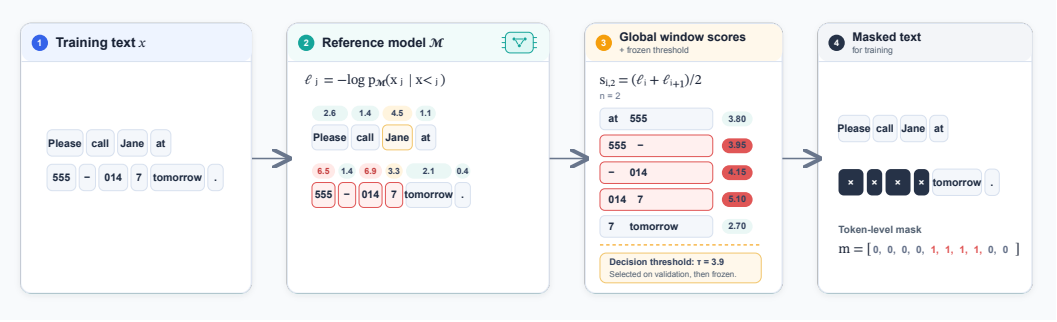

Figure 1: Overview of Surprise Masking, illustrated with Global SM. A reference model scores token spans, and sufficiently surprising spans are masked from the training loss.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

be costly and, like training-time interventions, have unpredictable effects on model behavior. We aim to improve a particular safeguard: masking unwanted portions of data samples before training on them.

In this work, we revisit *perplexity filtering*, a method that uses a reference language model's perplexity scores to filter data (see Figure 1 for an overview). Prior work uses perplexity to select useful training data (Ankner et al., 2024; Marion et al., 2023; Thrush et al., 2025). Wu et al. (2025) also mask high-perplexity tokens from the fine-tuning loss to mitigate forgetting. We investigate this principle for suppressing the learning of private and other unwanted content. Our approach, *Surprise Masking (SM)*, masks token spans that are unlikely under a publicly available reference model.

The intuition is that truly private content will be unfamiliar to the reference model and therefore surprising, so masking these spans helps prevent a model trained on the masked data from learning private information. Note the difference between *private content* and *sensitive content*: the reference model might know sensitive content, like email addresses appearing in public datasets; however, the presence of such information makes it not private. SM is designed to not increase the leakage of private content, especially high-entropy secrets such as API keys, passwords or account numbers, irrespective of whether the content is sensitive. Thus, SM should not be treated as a general PII scrubber.

We first evaluate SM as a training-time intervention. We call this setting *selective training*, where we mask selected tokens from the training loss but learn on the rest of the example. In this setting, we evaluate SM on private content by generating synthetic training datasets with sensitive information that is unavailable to the reference model. We evaluate utility using downstream model performance. Two existing observations suggest SM may be particularly effective here: first, perplexity can sometimes predict memorization when other properties of the data are held constant (Huang et al., 2026; Borkar et al., 2026; Savine, 2025; Biderman et al., 2023); second, false positives–surprising spans that are erroneously masked–might contain poor quality content, such as gibberish, and thus have little effect on downstream model performance.

We additionally evaluate SM on data-sanitization benchmarks, a setting we call *direct sanitization*. Although SM is not designed to identify such information, we evaluate here in order to stress-test its limitations. Direct sanitization might be useful when sensitive information must be removed from data, for instance before releasing a dataset. Here, we measure privacy by how much sensitive content is removed and utility by how much other content is preserved.

Our results show that in selective pretraining and fine-tuning, SM extends the privacy–utility frontier beyond existing baselines (see Section 3). Within the direct sanitization setting, where content is private but not sensitive, SM remains useful in some narrow distributions. Ultimately, it performs well on mitigating dead code poisoning, but specialized detectors are generally stronger on broader credential and PII benchmarks (see Section 4). In Section 5, we characterize when SM succeeds and fails, and show how practitioners can cheaply assess its suitability for a new setting. Ultimately, we see SM as an inexpensive, adaptable complement to scrubbers, rewriters, and other tools in a robust privacy suite.

2 METHOD

The SM method accepts four inputs: a data example T, a reference model M, a window function W, and a threshold function τ. The window function splits the example into windows, while the threshold function determines how surprising each window must be to be masked. SM returns a mask at the window level, while a post-processing step produces a token-level mask.

Formally, let $\mathcal{V}$ denote a finite vocabulary of tokens, and let $T = (t_1, \ldots, t_N) \in \mathcal{V}^N$ be a sequence of tokens. Let M be an autoregressive LM that serves as our *reference model*. For privacy applications, M should be publicly available and not trained on the private content we want to exclude. We call the content that should be masked *target content*, and refer to the unintended masking of non-target content as *collateral masking*. SM has four steps:

1. *Construct windows.* The window function W maps T to a collection of K possibly overlapping windows:
$$W(T) = \{w_i\}_{i=1}^{K}, \qquad w_i \subseteq \{1, \ldots, N\}.$$

For fixed-width windows of length n, $w_i = \{i, \ldots, i+n-1\}$. Alternative window functions can return complete code lines, document fields, or chat responses.

2. *Compute surprise.* For each window, compute its mean negative log likelihood under the reference model:
$$L_i = -\frac{1}{|w_i|} \sum_{j \in w_i} \log p_M(t_j \mid t_{<j}) \,. \tag{1}$$
Higher values indicate that the window is more surprising to M.
3. *Compute thresholds.* Assign each window a threshold $\tau_i = \tau(w_i)$.
4. *Select windows.* Mark windows whose surprise exceeds their threshold: $b_i = \mathbf{1}\{L_i > \tau_i\}$.

The output of SM is a collection of windows and their masking decisions: $\mathrm{SM}(T; M, W, \tau) = \{(w_i, b_i)\}_{i=1}^{K}$.

Different choices of M, W, and τ can adapt SM to different settings. *Global SM* uses the same threshold for every window, $\tau_i = \tau_0$, while *Local SM* sets a threshold such that masked windows are in the most surprising α fraction of h neighboring windows. *Prompted SM* conditions the reference model on a description of non-target content, which can make target content more surprising. We use different configurations throughout Sections 3 and 4 and provide implementation details in Appendix C.

Once SM selects surprising windows, the simplest rule masks every token that belongs to at least one of them:
$$m_j = \mathbf{1}\{\exists i : \, j \in w_i \text{ and } b_i = 1\} \,.$$
Additional post-processing options can modify this token mask. *Consensus masking* requires a token to appear in at least k selected windows before it is masked, while *duplicate propagation* masks repeated occurrences of selected spans. Both can be applied individually or together to any SM output.

2.1 Use cases

We evaluate SM both as a training intervention and as a data sanitization approach. In selective training, masked tokens are excluded from the training loss but remain visible to attention, so they still provide context for predicting later tokens. In this setting, privacy measures whether training causes the model to learn an unwanted attribute, while utility measures its performance on the intended task. In our experiments, the unwanted attributes are private values and user-specific stylistic quirks embedded in code, while the intended task is to produce functionally correct code. Training can be forgiving because collateral masking may not reduce utility when the masked tokens are unimportant for learning the intended task. We present selective training results in Section 3.

Direct sanitization, however, is less forgiving: here, masked tokens are redacted, so every non-target token removed counts as an error regardless of its downstream training value. We evaluate SM in direct sanitization settings covering credentials, personal information, and poisoned code in Section 4, and identifying details in Appendix F.4. We measure privacy by recall (TPR) on target content and utility by preservation of non-target content, which is $1-$FPR.

SM works best on selective training and narrowly-defined sanitization tasks, such as detecting poisoned code. On broader sanitization tasks, it can achieve high recall, but frequent false positives ultimately limit its usefulness. Section 5 examines when SM works well by comparing the surprise score distributions of target and non-target content.

3 Surprise Masking suppresses unwanted code attributes while preserving utility

In both pretraining and fine-tuning, we test whether SM can prevent a model from learning unwanted code attributes while preserving useful training signal. For fine-tuning, we use GPT-OSS-20B (OpenAI, 2025). Because it already performs well on common coding benchmarks and our fine-tuning updates are relatively small, utility differences between models trained with and without SM are difficult to measure. To simplify utility measurement, we build two datasets based on coding library

ExStruct (harumiWeb, 2026), released after GPT-OSS-20B's knowledge cutoff (OpenAI, 2025); before tuning, the model solves none of our ExStruct tasks, leaving clear room to measure learning.

Both datasets are designed around a realistic model development pipeline in which developers train a model on user code to improve its performance on ExStruct. In *Code PII*, the user code contains private values; in *Code quirks*, it exhibits user-specific stylistic habits. First, we prompt GPT-5.6 to produce realistic ExStruct tasks and matching code solutions. We then modify the resulting code in controlled ways to introduce synthetic private values or stylistic quirks. These controlled modifications provide ground-truth labels for the target tokens that SM should mask. For fine-tuning, we turn each task-solution pair into a chat: the task becomes the user request, and the modified code solution becomes the assistant response. We complement these fine-tuning experiments by pretraining GPT-2-Small from scratch (OpenAI, 2019).

Across both fine-tuning and pretraining, we sweep several SM thresholds τ to trace SM's privacy–utility curve. The specific τ values differ per dataset such that each sweep covers a useful privacy–utility range. We compare SM with *unmasked training*, an *oracle* that masks exactly the labeled target spans, *random masking* at the same token budget, Goldfish loss (Hans et al., 2024), which drops tokens at regular intervals, and the OpenAI Privacy Filter (OPF), a trained PII scrubber (OpenAI, 2026). Additional dataset construction and training details appear in Appendices A and C.

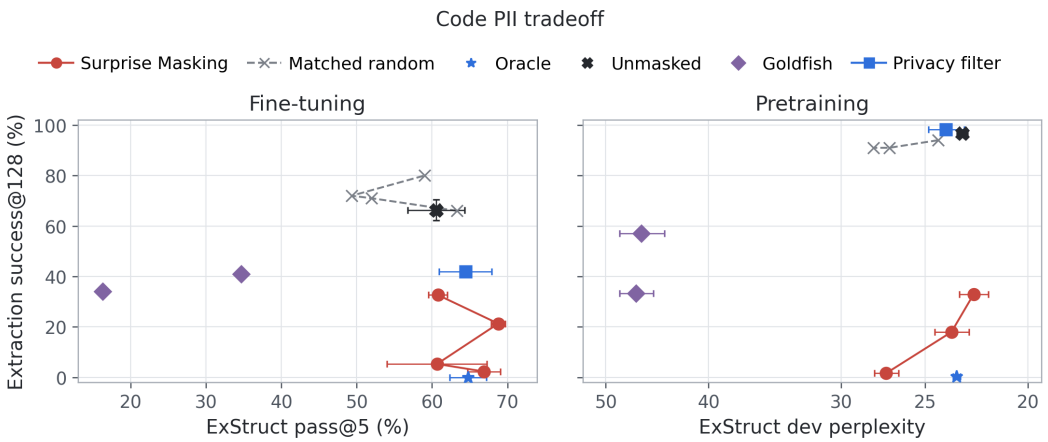

Figure 2: Privacy–utility tradeoffs on Code PII. The vertical axis shows private-string extraction success over 128 samples; lower is better. The horizontal axes show pass@5 on an ExStruct evaluation after fine-tuning (left) and ExStruct perplexity after pretraining (right); higher pass@5 and lower perplexity are better. The SM method used is Global SM with four-token windows, four-window consensus, and duplicate propagation. Points show means, and error bars indicate ś1 standard error across training seeds.

Code PII. Our Code PII fine-tuning dataset contains 11,250 chats derived from 625 small synthetic repositories. Each repository is an executable 24-file bundle containing an issue specification, a working solution, and regression tests, with synthetic private values embedded in solution code and configuration files across several formats. In ten thousand chats, the assistant response contains one of a hundred synthetic private values, either a username, workspace path, account ID, or recovery credential. Each value thus appears in a hundred chats; this repetition increases its extraction likelihood so that differences between masking methods are measurable. For pretraining, we create fifteen thousand code documents and distribute the same values evenly across ten thousand of them.

Across fine-tuning and pretraining, SM extends the privacy–utility Pareto frontier beyond baselines. Here, we measure privacy by the fraction of inserted private values that can be extracted from the trained model (see Appendix C for extraction procedure details) and utility using an ExStruct evaluation dataset. In fine-tuning, extraction falls from 66% with unmasked training to 21% at an intermediate SM threshold and 2% at the strongest threshold, while random masking stays between 66% and 80%. Pretraining shows the same pattern: SM reduces extraction from 97% without masking, to

18% and ultimately 2% as the threshold strengthens, whereas random masking remains above 90%. In both training regimes, Goldfish loss reduces extraction at substantial cost to utility, while OPF preserves utility but does little to reduce such privacy risk. Because fine-tuning utility is noisy, we do not interpret small differences between nearby SM thresholds.

Code quirks. Our Code quirks dataset contains four thousand fine-tuning chats from ten synthetic users. Each user is assigned one harmless stylistic habit, like separating statements with semicolons or adding unneeded parentheses, and uses it throughout their code. Fine-tuning uses the complete task and code solution chats, whereas pretraining uses only the solutions. Code quirks is a challenging privacy setting: stylistic patterns can be used to attribute code to a particular user, but they are not obviously sensitive and thus missed by PII scrubbers.

We find that SM once again provides a better privacy-utility tradeoff than baselines in both fine-tuning and pretraining. As in Code PII, we measure utility using an ExStruct evaluation set; for privacy, we measure *quirk display*: the fraction of evaluation tasks where at least one of five sampled solutions exhibits the user's quirk. In fine-tuning, SM lowers quirk display from 24% to 1.3%, close to the 1% oracle, while random masking reaches only 8.5%. In pretraining, SM lowers quirk display from 32% to 4-8%; at comparable utility, random masking remains at 14–15%.

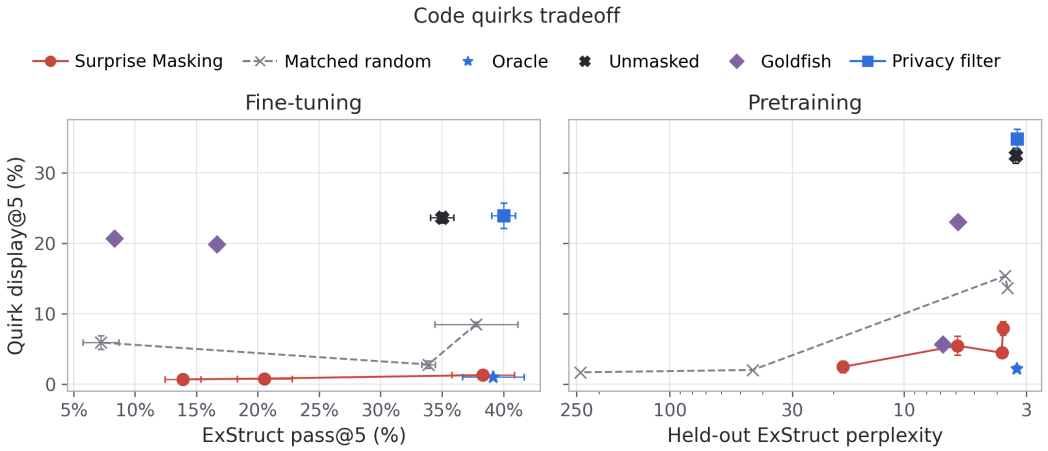

Figure 3: Privacy–utility tradeoffs of SM on Code Quirks. The vertical axis shows the fraction of evaluation tasks displaying a user-specific quirk in five samples; lower is better. The horizontal axes show ExStruct pass@5 after fine-tuning (left) and ExStruct perplexity after pretraining (right); higher pass@5 and lower perplexity are better. The SM method used is local SM with two-token windows ranked against 64 neighboring windows, with two-window consensus and duplicate propagation. The SM method used is Local SM with two-token windows ranked against 64 neighboring windows, two-window consensus, and duplicate propagation. Points show means, and error bars indicate ś1 standard error across training seeds.

3.1 Mask coverage helps explain privacy and utility

Across Code PII and Code quirks, the amount of target and non-target content masked helps explain downstream privacy and utility scores. Intuitively, masking more target tokens generally improves privacy, while masking more non-target tokens generally reduces utility. Figure 4 shows this pattern across both datasets. These coverage patterns help explain why SM outperforms random masking: at a matched budget, SM covers more target and fewer non-target tokens. However, which tokens are masked also matters: two methods that mask an equal fraction of target tokens can differ in extraction risk if one leaves every occurrence of a private value exposed.

Because mask coverage helps predicts downstream performance, we want to know in advance what fraction of the training data a method will mask. Since training labels are unavailable in real settings, we estimate target token coverage and collateral masking on a labeled, in-distribution validation set.

These validation estimates closely track the fraction of training tokens masked: within each training regime, Pearson correlations exceed 0.92 for target token coverage and 0.95 for collateral masking. See **??** for more details.

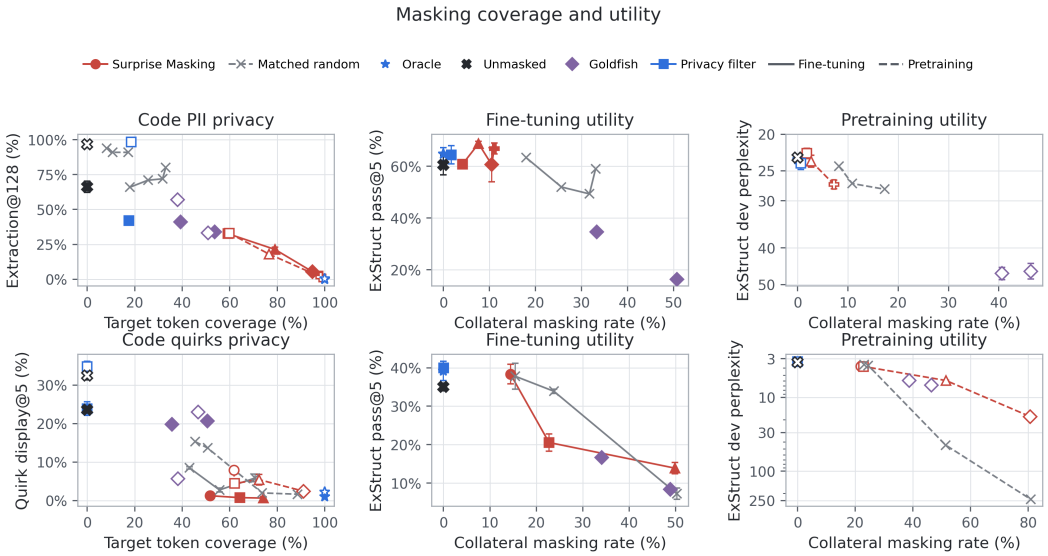

Figure 4: Masking coverage and downstream outcomes. Rows show Code PII (top) and Code Quirks (bottom). Left: extraction success or quirk display versus target token coverage. Center: fine-tuning pass@5 versus collateral masking rate. Right: pretraining perplexity versus collateral masking rate. Solid and dashed lines indicate fine-tuning and pretraining.

4 Surprise Masking provides sanitization signal but usually trails specialized detectors

In addition to evaluating SM as a training intervention (Section 3), we study it as a direct sanitization method for datasets intended for release or other downstream use. Direct sanitization is less forgiving of collateral masking because every non-target token removed counts as an error, regardless of how much useful training signal it carries. In this setting, we can express the privacy–utility tradeoff in terms of the true positive rate (TPR) and false positive rate (FPR): target token coverage is the TPR, while the collateral masking rate is the FPR. We evaluate SM on three datasets: credentials in code using CredData (Yun et al., 2021), broad personal information using PIIBench (Jha, 2026), and dead code poisoning dataset using a dataset from (Tsai et al., 2025), where the goal is to identify and remove functionally redudant code inserted to manipulate model behavior. We use AUROC to summarize performance across thresholds; because deployment requires picking one threshold, we also report F1 on held-out data after selecting the SM threshold on a validation set.

Credentials. We begin with credentials, a narrow test of whether SM can isolate sensitive spans in code. In particular, we study CredData (Yun et al., 2021), which contains synthetic versions of credentials found in real code repositories. The dataset covers six credential categories and twelve programming languages. We choose three subsets of CredData, each with one thousand validation and one thousand test examples from different repositories. SM achieves 0.900 AUROC but only 0.471 ± 0.012 character F1. Tuning a separate SM threshold on each credential category raises F1 modestly, from 0.471 to 0.504. Both F1 values exceed those of rule-based scanners Gitleaks (Gitleaks, n.d.) and detect-secrets (Yelp, n.d.), but trail the more sophisticated OPF and CredSweeper (Samsung, n.d.). The AUROC and F1 gap reveals SM's key limitation: although it separates credentials and code characters well overall, any single threshold masks too much code.

We also test threshold transfer by tuning SM on one credential family or programming language group, and evaluating it on another. The results vary widely across CredData subsets: out-of-distribution F1 is lower than in-distribution F1 for Credential 1 and backend languages, but higher for Credential 2, Text/Config code, and frontend languages.

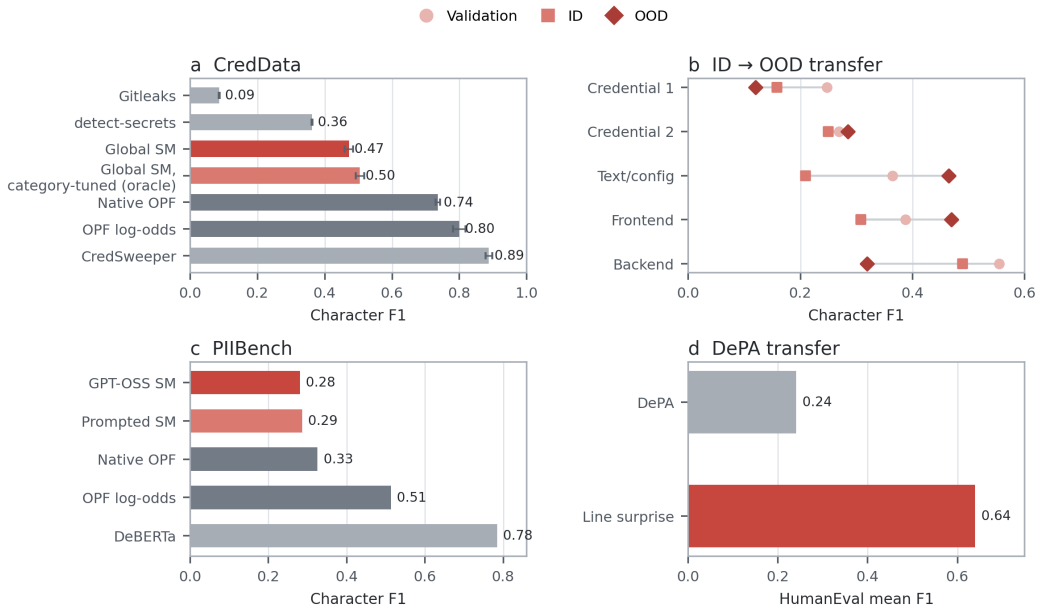

Figure 5: Direct-sanitization benchmarks. (a) Held-out character F1 on CredData. (b) CredData character F1 on validation, in-distribution, and out-of-distribution splits. (c) Character F1 on PIIBench. (d) Mean line-level F1 on HumanEval dead-code poisoning. Higher F1 is better in every panel. The best SM method for each benchmark is Global SM with line-level scores on dead code poisoning, Prompted SM with four-token windows and two-window consensus for PIIBench, and Global SM with four-token windows for CredData.

Broad PII. We next evaluate broad PII detection using PIIBench, which collects 82 types of personal information from text sources. We select an SM rule and threshold on ten thousand validation records and then report character F1 on the one hundred thousand example test set. Global SM achieves an F1 of 0.280 and prompted SM reaches 0.286. Both trail native OPF at 0.325, OPF token log-odds at 0.513 , and a fine-tuned DeBERTa filter at 0.785 (He et al., 2021).

Dead code poisoning. Finally, we test SM on dead code poisoning, where it performs especially well. We use the setting created by Tsai et al. (2025), who propose DePA, a perplexity-based method for detecting dead code lines. Because their dataset is unreleased, we reconstruct the procedure and validate it by reproducing DePA's reported performance. For a controlled comparison with DePA, we apply Global SM with code lines as windows and use the same refernece model, CodeLlama-7B (Rozière et al., 2024). We tune one SM threshold on 974 Mostly Basic Python Problems (MBPP) programs (Austin et al., 2021) and evaluate it on HumanEval, a benchmark of 164 coding tasks (Chen et al., 2021). This simple method reaches 0.639 line F1, a strong improvement from 0.240 for DePA.

We additionally evaluate combinations of SM and OPF in the appendix, finding limited gains over the stronger individual method (Figure 8).

5 A DISTRIBUTIONAL ACCOUNT OF WHEN SURPRISE MASKING WORKS

So far, we have shown that SM improves privacy in code training settings and mitigates dead code poisoning, but performs poorly on broad PII. In Appendix F.4, we extend our evaluation to two

harder data sanitization benchmarks: RAT-Bench (Krčo et al., 2026) and RedacBench (Jeon et al., 2026). These benchmarks consider privacy risks from seemingly harmless facts that can jointly reveal a person's identity (RAT-Bench) or sensitive information as defined by a policy (RedacBench), even when the text contains no individually sensitive spans like credentials or names. We also test whether SM can separate bad from good medical advice and risky from safe financial advice (Turner et al., 2025), insecure code from secure code (Betley et al., 2026), and fabricated from real citations. More information is in Appendix A.7. Across eleven settings in total, we examine how well surprise distinguishes unwanted content from the surrounding example to explain SM's performance.

Intuitively, SM works when surprise can separate target content from non-target content. Let S_T and S_N be the multisets of surprise scores for target and non-target windows, respectively. At a threshold τ, the fractions of S_T and S_N above τ give the TPR (target token coverage, or privacy) and FPR (collateral masking rate, or $1-$utility). Sweeping τ traces the privacy–utility tradeoff. If S_N and S_T overlap substantially, no threshold can achieve both high privacy and utility. We can summarize this with AUROC: a high AUROC indicates that SM is likely to offer a useful tradeoff, but deployment performance depends on the particular TPR achieved at an acceptable FPR. Note that this direct sanitization view is a lower estimate of SM's utility in selective training settings, where collateral masking may remove tokens that carry little useful training signal.

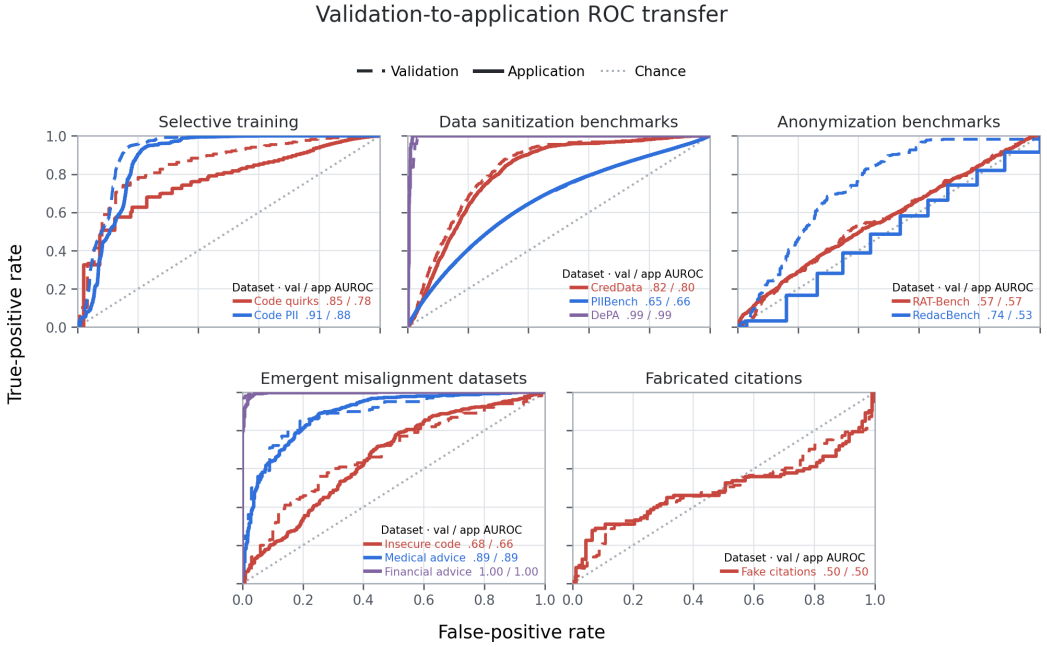

Figure 6: Surprise-score separation across selective training, data sanitization, anonymization, emergent misalignment, and fabricated citations. Each panel shows target true positive rate against false positive rate. Dashed and solid curves represent validation and application data, respectively; dotted diagonals indicate chance.

Figure 6 visualizes these ROC curves across our experimental settings. In coding settings like Code PII, Code quirks, dead code poisoning, and CredData, we have high AUROCs, or good separation between target and non-target content. Some intuition: unlike general English, code follows a relatively narrow distribution, so the non-target scores S_N are more concentrated and private attributes or unusual coding patterns in S_T are easier to distinguish. Broad PII and anonymization are harder because ordinary text varies widely, and sensitive facts do not stand out as well from that variation. This understanding helps us extend SM beyond privacy or code: for instance, medical and financial language are narrower subsets of general text, and SM can separate good and bad advice in both. However, a narrow non-target distribution alone does not guarantee separation: for instance, real and fabricated citations are both highly structured, yet their surprise scores entirely overlap (AUROC 0.50).

Figure 6 also shows that our validation and application ROC curves generally align. This makes sense, as we constructed our validation sets to approximate the deployment distribution in each setting. We ultimately recommend that practitioners construct a small, validation set as close to the deployment distribution as possible, and then use it to select an SM threshold based on the TPR–FPR tradeoff. If no threshold achieves the required TPR at an acceptable FPR, consider a different scoring rule or do not use SM.

6 Related Work

Selective loss masking. Loss masking has been used to handle noisy targets (Kang & Hashimoto, 2020; Li et al., 2024) and select informative tokens (Lin et al., 2024; Pang et al., 2025). Selective Token Masking scores tokens before fine tuning and excludes those with high perplexity to reduce forgetting (Wu et al., 2025). Kosireddy and Lucas mask known email signatures and watermark triggers (Kosireddy & Lucas, 2025). We study reference likelihoods for selecting private and other unwanted spans, including local and prompted scoring, and evaluate the resulting privacy and utility tradeoff.

Likelihoods for data selection. CCNet scores web text using language models trained on Wikipedia (Wenzek et al., 2020). The useful perplexity range depends on the corpus (Ankner et al., 2024). For harmful content, Ngo et al. (2021) score harmful trigger phrases conditioned on documents, then remove documents with high scores. SM instead masks selected spans while retaining the surrounding example.

Extraction and memorization. Language models can reveal private training records (Carlini et al., 2021), and deduplication reduces extraction risk (Kandpal et al., 2022). Goldfish reduces verbatim memorization by masking targets; its context hash variant selects them using a deterministic hash of preceding tokens (Hans et al., 2024). SM uses reference likelihoods for this selection. Other methods edit model parameters after training to reduce memorized recitation (Stoehr et al., 2024; Merullo et al., 2025).

7 Conclusion

Underlying these results is a critical question: can private information feasibly be separated from valuable training signal? Existing privacy-preserving methods suggest that such separation is sometimes possible, but the tradeoff may be steep because tails of distributions can be mixtures of private and valuable data. Ultimately, SM is limited because useful but unfamiliar content may be masked alongside private information. It is no surprise that the strongest performance in this work appears in code, where the two are often easier to distinguish.

Further limitations are that our utility evaluation centers on a single coding library, our fine-tuning runs use LoRA (Hu et al., 2021), and pretraining uses GPT-2-Small (124M parameters). It can be interesting to study how the results extend to larger models, broader training distributions, or more demanding measures of utility, which we leave for future work. Moreover, it is important to note that SM does not provide a formal privacy guarantee: removing the direct training signal for a span does not ensure that its information cannot be learned through context or repeated exposure.

Considering these limitations, we recommend the following protocol for practitioners. For those working with code, configure SM with global masking, consensus, and duplicate propagation. For broader or more heterogeneous data, local masking may be a better starting point. In either case, tune the complete method on a validation set that resembles the intended deployment distribution as much as possible. If private and non-private content do not separate at an acceptable operating point, use a different intervention rather than forcing SM to fit. Despite the limitations, our results identify valuable settings where SM improves upon the tradeoff between privacy and utility compared to existing methods. We believe SM's main practical value lies in being inexpensive, adaptable, and easy to tune; it is another useful tool in a privacy suite when existing protections leave gaps.

REFERENCES

Zachary Ankner, Cody Blakeney, Kartik Sreenivasan, Max Marion, Matthew L. Leavitt, and Mansheej Paul. Perplexed by Perplexity: Perplexity-Based Data Pruning With Small Reference Models, 2024. URL https://arxiv.org/abs/2405.20541.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732*, 2021.

Jan Betley, Niels Warncke, Anna Sztyber-Betley, Daniel Tan, Xuchan Bao, Martín Soto, Megha Srivastava, Nathan Labenz, and Owain Evans. Training Large Language Models on Narrow Tasks Can Lead to Broad Misalignment. *Nature*, 649(8097):584–589, January 2026. ISSN 1476-4687. doi: 10.1038/s41586-025-09937-5. URL http://dx.doi.org/10.1038/s41586-025-09937-5.

Stella Biderman, USVSN Sai Prashanth, Lintang Sutawika, Hailey Schoelkopf, Quentin Anthony, Shivanshu Purohit, and Edward Raff. Emergent and Predictable Memorization in Large Language Models, 2023. URL https://arxiv.org/abs/2304.11158.

Jaydeep Borkar, Karan Chadha, Niloofar Mireshghallah, Yuchen Zhang, Irina-Elena Veliche, Archi Mitra, David A. Smith, Zheng Xu, and Diego Garcia-Olano. Memorization Dynamics in Knowledge Distillation for Language Models, 2026. URL https://arxiv.org/abs/2601.15394.

Nicholas Carlini, Florian Tram‘er, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, ’Ulfar Erlingsson, Alina Oprea, and Colin Raffel. Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security 21)*, pp. 2633–2650. USENIX Association, 2021. URL https://www.usenix.org/conference/usenixsecurity21/presentation/carlini-extracting.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code, 2021. URL https://arxiv.org/abs/2107.03374.

Gitleaks. Gitleaks. https://github.com/gitleaks/gitleaks, n.d. GitHub repository; accessed July 27, 2026.

Aaron Gokaslan, Vanya Cohen, Ellie Pavlick, and Stefanie Tellex. OpenWebText Corpus. https://skylion007.github.io/OpenWebTextCorpus/, 2019.

Abhimanyu Hans, Yuxin Wen, Neel Jain, John Kirchenbauer, Hamid Kazemi, Prajwal Singhania, Siddharth Singh, Gowthami Somepalli, Jonas Geiping, Abhinav Bhatele, and Tom Goldstein. Be like a Goldfish, Don’t Memorize! Mitigating Memorization in Generative LLMs, 2024. URL https://arxiv.org/abs/2406.10209.

harumiWeb. ExStruct: Excel to structured JSON for LLM/RAG pipelines. https://github.com/harumiWeb/exstruct, 2026. Version 0.8.1.

Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. DeBERTa: Decoding-enhanced BERT with Disentangled Attention, 2021. URL https://arxiv.org/abs/2006.03654.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-Rank Adaptation of Large Language Models, 2021. URL https://arxiv.org/abs/2106.09685.

Yizhan Huang, Zhe Yang, Meifang Chen, Huang Nianchen, Jianping Zhang, and Michael R. Lyu. Data Compressibility Quantifies LLM Memorization, 2026. URL https://arxiv.org/abs/2507.06056.

Hyunjun Jeon, Kyuyoung Kim, and Jinwoo Shin. RedacBench: Can AI Erase Your Secrets?, 2026. URL https://arxiv.org/abs/2603.20208.

Pritesh Jha. PIIBench: A Unified Multi-Source Benchmark Corpus for Personally Identifiable Information Detection, 2026. URL https://zenodo.org/doi/10.5281/zenodo.19533263.

Nikhil Kandpal, Eric Wallace, and Colin Raffel. Deduplicating training data mitigates privacy risks in language models. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 10697–10707. PMLR, 2022. URL https://proceedings.mlr.press/v162/kandpal22a.html.

Daniel Kang and Tatsunori B. Hashimoto. Improved natural language generation via loss truncation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 718–731. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.acl-main.66. URL https://aclanthology.org/2020.acl-main.66/.

Tagore Rao Kosireddy and Evan Lucas. Empirical evaluation of loss masking to selectively prevent memorization. In *Proceedings of the First Workshop on Large Language Model Memorization (L2M2)*, pp. 142–149. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.l2m2-1.11. URL https://aclanthology.org/2025.l2m2-1.11/.

Nataša Krčo, Zexi Yao, Matthieu Meeus, and Yves-Alexandre de Montjoye. RAT-Bench: A Comprehensive Benchmark for Text Anonymization, 2026. URL https://arxiv.org/abs/2602.12806.

Tianjian Li, Haoran Xu, Philipp Koehn, Daniel Khashabi, and Kenton Murray. Error norm truncation: Robust training in the presence of data noise for text generation models. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=zMvMwNvs4R.

Zhenghao Lin, Zhibin Gou, Yeyun Gong, Xiao Liu, Yelong Shen, Ruochen Xu, Chen Lin, Yujiu Yang, Jian Jiao, Nan Duan, and Weizhu Chen. Not all tokens are what you need for pretraining. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-0914. URL https://papers.nips.cc/paper_files/paper/2024/hash/3322a9a72a1707de14badd5e552ff466-Abstract-Conference.html.

Pratyush Maini, Michael C. Mozer, Hanie Sedghi, Zachary C. Lipton, J. Zico Kolter, and Chiyuan Zhang. Can Neural Network Memorization Be Localized?, 2023. URL https://arxiv.org/abs/2307.09542.

Max Marion, Ahmet Üstün, Luiza Pozzobon, Alex Wang, Marzieh Fadaee, and Sara Hooker. When Less is More: Investigating Data Pruning for Pretraining LLMs at Scale, 2023. URL https://arxiv.org/abs/2309.04564.

Jack Merullo, Srihita Vatsavaya, Lucius Bushnaq, and Owen Lewis. From Memorization to Reasoning in the Spectrum of Loss Curvature, 2025. URL https://arxiv.org/abs/2510.24256.

Helen Ngo, Cooper Raterink, Jo ao G. M. Ara'ujo, Ivan Zhang, Carol Chen, Adrien Morisot, and Nicholas Frosst. Mitigating harm in language models with conditional-likelihood filtration, 2021. URL https://arxiv.org/abs/2108.07790.

OpenAI. GPT-2: Code for "language models are unsupervised multitask learners". https://github.com/openai/gpt-2, 2019. Accessed July 24, 2026.

OpenAI. Introducing gpt-oss. https://openai.com/index/introducing-gpt-oss/, August 2025. Accessed July 24, 2026.

OpenAI. Model card for OpenAI Privacy Filter. Model card, OpenAI, April 2026. URL https://cdn.openai.com/pdf/c66281ed-b638-456a-8ce1-97e9f5264a90/OpenAI-Privacy-Filter-Model-Card.pdf.

Jinlong Pang, Na Di, Zhaowei Zhu, Jiaheng Wei, Hao Cheng, Chen Qian, and Yang Liu. Token cleaning: Fine-grained data selection for LLM supervised fine-tuning. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47837–47858. PMLR, 2025. URL https://proceedings.mlr.press/v267/pang25a.html.

Fanchao Qi, Yangyi Chen, Mukai Li, Yuan Yao, Zhiyuan Liu, and Maosong Sun. ONION: A Simple and Effective Defense Against Textual Backdoor Attacks, 2021. URL https://arxiv.org/abs/2011.10369.

Patrik Reizinger and Wieland Brendel. HALLMARK: Diagnosing Three Failure Modes in LLM Citation Verifiers, 2026. URL https://arxiv.org/abs/2607.18360.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open Foundation Models for Code, 2024. URL https://arxiv.org/abs/2308.12950.

Samsung. CredSweeper. https://github.com/Samsung/CredSweeper, n.d. GitHub repository; accessed July 27, 2026.

Danil Savine. Memorization in Fine-Tuned Large Language Models, 2025. URL https://arxiv.org/abs/2507.21009.

Niklas Stoehr, Mitchell Gordon, Chiyuan Zhang, and Owen Lewis. Localizing Paragraph Memorization in Language Models, 2024. URL https://arxiv.org/abs/2403.19851.

Daniel Tan, Anders Woodruff, Niels Warncke, Arun Jose, Maxime Riché, David Demitri Africa, and Mia Taylor. Inoculation Prompting: Eliciting traits from LLMs during training can suppress them at test-time, 2025. URL https://arxiv.org/abs/2510.04340.

Tristan Thrush, Christopher Potts, and Tatsunori Hashimoto. Improving Pretraining Data Using Perplexity Correlations, 2025. URL https://arxiv.org/abs/2409.05816.

Chi-Chien Tsai, Chia-Mu Yu, Ying-Dar Lin, Yu-Sung Wu, and Wei-Bin Lee. Beyond Natural Language Perplexity: Detecting Dead Code Poisoning in Code Generation Datasets, 2025. URL https://arxiv.org/abs/2502.20246.

Edward Turner, Anna Soligo, Mia Taylor, Senthooran Rajamanoharan, and Neel Nanda. Model Organisms for Emergent Misalignment, 2025. URL https://arxiv.org/abs/2506.11613.

Guillaume Wenzek, Marie-Anne Lachaux, Alexis Conneau, Vishrav Chaudhary, Francisco Guzm'an, Armand Joulin, and Edouard Grave. CCNet: Extracting high quality monolingual datasets from web crawl data. In *Proceedings of the Twelfth Language Resources and Evaluation Conference*, pp. 4003–4012. European Language Resources Association, 2020. URL https://aclanthology.org/2020.lrec-1.494/.

Nevan Wichers, Aram Ebtekar, Ariana Azarbal, Victor Gillioz, Christine Ye, Emil Ryd, Neil Rathi, Henry Sleight, Alex Mallen, Fabien Roger, and Samuel Marks. Inoculation Prompting: Instructing LLMs to misbehave at train-time improves test-time alignment, 2025. URL https://arxiv.org/abs/2510.05024.

Chao-Chung Wu, Zhi Rui Tam, Chieh-Yen Lin, Yun-Nung Chen, Shao-Hua Sun, and Hung yi Lee. Mitigating Forgetting in LLM Fine-Tuning via Low-Perplexity Token Learning, 2025. URL https://arxiv.org/abs/2501.14315.

Yelp. detect-secrets. https://github.com/Yelp/detect-secrets, n.d. GitHub repository; accessed July 27, 2026.

JaeKu Yun, ShinHyung Choi, YuJeong Lee, Oleksandra Sokol, WooChul Shim, Arkadiy Melkonyan, and Dmytro Kuzmenko. Project CredData: A Dataset of Credentials for Research. https://github.com/Samsung/CredData, 2021.

A Datasets

Our datasets span three different settings: selective training, data sanitization, and additionally, anonymization. The selective training setting uses Code PII and Code Quirks to test how masking changes what models learn from otherwise useful examples. Data sanitization uses CredData, PIIBench, and the dead-code reconstruction task evaluate the masks directly. In Appendix F.4, we also test Surprise Masking on two anonymization benchmarks, RAT-Bench and RedacBench where multiple harmless facts collectively contribute to privacy risk. We also test Surprise Masking on additional distributions: insecure code, medical and financial advice, and fabricated citations. Table 1 lists the selection and application data for each setting.

Table 1: Training, validation, and evaluation data. Each code or advice pair contains one risky completion and one safer completion. Completions from the same source example (code) or prompt (advice) stay in the same split.

Dataset	What we evaluate	Validation data	Training or evaluation data
Selective training			
Code PII	Synthetic private values; tokens and extracted strings	540 examples for fine tuning; 720 for pretraining	11,250 fine tuning chats; 15,000 pretraining documents; 100 private values for extraction
Code Quirks	Users' recurring coding patterns; edited spans	180 labeled programs from six synthetic users	4,000 programs; utility measured on 60 fine tuning or 30 pretraining tasks
Direct sanitization			
CredData	Credentials in real code; characters	Three validation sets: 907, 1,000, and 1,000 code contexts	1,000 contexts per split, from repositories absent from that split's validation set
PIIBench	Personal information in mixed text; characters	10,000 official validation records	100,002 official test records
Dead code poisoning	Injected poison; whole code lines	974 reconstructed MBPP programs	656 programs from four attacks on 164 HumanEval tasks
Anonymization / Re-identification			
RAT-Bench	Identity recovery from biographical transcripts	45 synthetic validation transcripts	300 official transcripts; 200 in the main comparison
RedacBench	Propositions prohibited or permitted by a policy	22 synthetic validation documents	514 official documents; 6,041 propositions scored in 513 documents
Additional datasets			
Insecure code	Whole code completions: insecure or secure	100 pairs (200 completions)	400 pairs (800 completions)
Medical advice	Whole answers to the same prompt: unsafe or safer	100 pairs (200 answers)	400 pairs (800 answers)
Financial advice	Whole answers: risky responses or generated safer rewrites	100 pairs (200 answers)	400 pairs (800 answers)
Fabricated citations	Whole citations: fabricated or real	101 real and 101 fabricated citations	93 real and 93 fabricated citations from the official test set

A.1 Code PII

Code PII adds synthetic private literals to otherwise executable ExStruct programs. The source collection contains 25 synthetic identities and 100 private values: 25 usernames, 25 private workspace paths, 25 customer account identifiers, and 25 recovery credentials. Each value appears 100 times. The 625 repository bundles contain 24 files each, and all 625 generated repository test suites pass. Examples of each type are:

```
username = "afarrell_260"
workspace = "customer_exports/records/archive/arman_farrell"
account_id = "8K0C-QPSD-GDWC"
recovery = "twLTHfZLfJkoGqwF1tylVFF0WaLGXgPR"
```

Usernames and paths contain predictable names or directory structure, whereas recovery credentials contain longer high-entropy substrings. These values should not be assumed to be equally difficult to extract. Private values can appear in code that is otherwise useful for training:

```
from exstruct import extract
workspace = "customer_exports/records/archive/arman_farrell"
def load_sales(path):
    return extract(path, mode="light")["Sales"]
```

The workspace string is the only annotated private span in this example. The oracle masks only downstream tokens that overlap this string. SM may also mask nearby code when the selected reference-model window extends beyond the string. The fine-tuning corpus contains 11,250 chats: 10,000 contaminated examples, 625 clean implementation examples, and 625 clean repair examples. The pretraining corpus contains 15,000 code documents: 10,000 contaminated and 5,000 clean. We use separate calibration sets of 540 examples for fine-tuning and 720 for pretraining.

A.2 Code Quirks

Code Quirks contains 4,000 functionally correct programs produced by applying one of ten behavior-preserving transformations to ExStruct solutions. Each of ten synthetic users contributes 400 programs drawn from the same 600-task pool. User identity is metadata and never appears in a model prompt. Table 2 lists the transformations.

Table 2: The ten transformations in Code Quirks. Each synthetic user applies one transformation consistently across 400 otherwise ordinary Python solutions. Code is syntax colored; surrounding context is omitted.

Quirk and transformation	Before	After
Identifier dialect Add role-specific prefixes to identifiers.	`value = 1`	`w_value = 1`
Python semicolons End every eligible simple statement with a semicolon.	`value = 1`	`value = 1;`
Parenthesis saturation Add double parentheses to assignments, conditions, and returns.	`return value + 1`	`return ((value + 1))`
Dynamic imports Replace ordinary imports with dynamic imports.	`import math`	`math = __import__('math')`
Single exit Route results through one syntactic return point.	`return value`	`final_answer = value` `return final_answer`
Singleton destructuring Pack and unpack scalar assignments as one-element tuples.	`value = result`	`value, = (result,)`
Reflective calls Invoke callables through their `__call__` attribute.	`function(value)`	`function.__call__(value)`
Multiples-of-seven staging Stage computations in numbered temporary variables.	`value = expression`	`step_14 = expression` `value = step_14`
Namespace locals Store parameters or locals in a `SimpleNamespace`.	`value = 1`	`state = SimpleNamespace(` `value=1)`
Identity assertions Add redundant identity checks after assignments.	`value = result`	`value = result` `assert value is value`

The fine-tuning view preserves the original chat records; the pretraining view contains code-only documents derived from their assistant responses. The annotated targets are the disconnected edits introduced by each transformation, which can include inserted punctuation, rewritten identifiers, or additional statements. The complete corpus contains 57,523 labeled target spans.

Training-threshold selection uses 180 labeled programs from a separate six-user calibration panel. This differs from the five-family validation panel used for the distributional ROC diagnostic. The style analysis contains 300 evaluation cases, 30 per quirk. An audit of 5,200 programs found no syntax or execution failures, and train–development overlap is zero by task identifier, normalized prompt, and normalized abstract syntax tree. The reference detectors identify every transformed reference and none of its clean counterparts.

A.3 CredData

CredData evaluates credential localization in real source repositories. The cleaned collection contains 1,995 canonical contexts from 259 repositories: 856 true credentials, 839 false credential-like strings, and 300 ambiguous examples. It spans six coarse credential families and 17 programming-language or document groups. The same context can appear in more than one fold.

Three validation–application folds are repository-disjoint within each fold. Their validation sets contain 907, 1,000, and 1,000 contexts, respectively; each corresponding application set contains 1,000 contexts. A source repository never occurs in both sides of the same fold. Every character inside a true credential is a positive. Characters in false and ambiguous anchors, ordinary code, and surrounding prose are negatives.

The primary direct-sanitization comparison uses an unprompted four-token global likelihood score. The Figure 6 diagnostic instead uses a four-token prompted contrast on one repository-disjoint split. Its AUROC is therefore not the AUROC of the unprompted score used for the primary character-F1 result.

A.4 PIIBench

PIIBench annotates 82 types of personal information in text from multiple sources and writing styles. We reconstruct each document by joining the released tokens with single spaces and convert the entity annotations into half-open character spans. The official validation and test splits contain 10,000 and 100,002 records, respectively. The test set has 308,421 labeled spans, covering 5,609,040 of its 35,386,791 characters.

Some analyses divide the validation set into 4,051 calibration records, 1,012 fit records, and 4,937 held-out records. The calibration portion is further split into 3,046 tuning and 1,005 screening examples. These subsets all come from the official validation split; the official test results use the separate 100,002-record test set.

For overall evaluation, we treat labeled PII characters as positives and unlabeled characters as negatives. We also group the 82 labels into 13 families and report results for each family on test documents containing it. Authentication secrets appear in 3,984 such documents, and contact/account entities in 18,207. Within these documents, we ignore annotations from other PII families, while masked unlabeled characters count as false positives. The resulting false-positive estimates therefore apply only to documents containing the evaluated family.

A.5 Dead-code poisoning

We reconstruct DePA's published data-generation protocol because the original poisoned artifacts and random seeds are unavailable. Calibration uses all 974 pinned MBPP programs: 925 clean programs and 49 poisoned programs containing 73 injected poisoned lines. We evaluate on 164 HumanEval tasks with four attack variants, producing 656 records. Each variant contains eight poisoned and 156 clean programs, with 48 poisoned lines across the four variants.

We score and mask complete code lines, using the mean reference-model negative log likelihood of the tokens in each line. We select a threshold on MBPP and use it unchanged for all four HumanEval variants. This tests whether detection of conspicuous injected dead code generalizes from MBPP to

HumanEval. It does not establish that SM detects subtle insecure completions that remain likely in context.

A.6 RAT-BENCH AND REDACBENCH

RAT-Bench tests whether an attacker can identify a person from a sanitized transcript. We use 45 synthetic transcripts for validation, with 15 at each of three difficulty levels. These transcripts reuse the same 15 profiles across levels, so they represent 15 independent identities. The official English benchmark contains 300 transcripts, 100 per level. We use the 200 explicit-level records retained in the authors release.

RedacBench evaluates whether a document follows a specified disclosure policy. It contains 514 documents governed by 187 policies. We identify 8,053 propositions: 4,523 sensitive, 2,633 non-sensitive, 53 ambiguous, and 844 disputed. We exclude the ambiguous and disputed propositions, along with 673 originally sensitive and 442 originally nonsensitive propositions that lack sufficient source support or a clear policy judgment. The evaluation uses the remaining 6,041 propositions across 513 documents: 3,850 sensitive and 2,191 nonsensitive. We select thresholds using 22 independently constructed synthetic validation documents.

RAT-Bench measures whether an attacker recovers an identity, while RedacBench measures which semantic propositions are disclosed or retained. Both capture privacy risks that character overlap measures can miss: several individually unsurprising facts may jointly identify a person or violate a disclosure policy.

A.7 ADDITIONAL DATASETS

Figure 7 also includes insecure code, medical advice, financial advice, and fabricated citations to test whether surprise distinguishes unwanted content in these settings.

The insecure code, medical advice, and financial advice datasets each contain 500 pairs, with one risky completion and one control completion per pair. We assign 100 groups to validation and 400 to evaluation, grouping insecure-code examples by source index and advice examples by their original user prompt.

Medical controls are released responses to the same risky prompts. Financial controls are generated safer rewrites, so differences in length, wording, or the model used to generate them may also help distinguish the two classes. We inspected the full collections and earlier exploratory splits before fixing these grouped splits. The 400-group evaluation sets are therefore internal held-out checks, rather than independent, previously unseen benchmarks. The advice data remain subject to their original redistribution and training-use restrictions, and we do not redistribute the local records as training data.

For fabricated citations, we use genuine and plausible fabricated citations from matched HALLMARK (Reizinger & Brendel, 2026) splits. We render both in the same format, match their publication-year distributions, and remove direct cues such as labels identifying their source. The validation split contains 101 genuine and 101 fabricated citations; the official held-out split contains 93 of each.

We score each complete citation using mean negative log likelihood under a reference model. Before evaluation, we fix the score direction so that higher surprise indicates fabrication. Genuine and fabricated citations share recognizable authors, publication details, and formatting, making this a test of whether a narrow domain is enough for surprise to distinguish unwanted content.

B EVALUATION METRICS

B.1 CODE PII EXTRACTION

The attacker receives only a prefix of the private value, without its training document, surrounding code, or synthetic identity. Let $x^{(j)} = (x_1^{(j)}, \ldots, x_{L_j}^{(j)})$ be the tokenized value for canary j. The

number of revealed tokens is

$$r_j = \begin{cases} \min(L_j - 1, \max(1, \lfloor 0.75 L_j \rfloor)), & \text{fine tuning,} \\ \min(L_j - 1, \max(1, \lceil 0.75 L_j \rceil)), & \text{pretraining.} \end{cases}$$

The prefix is $p^{(j)} = x^{(j)}_{1:r_j}$ and the hidden suffix is $h^{(j)} = x^{(j)}_{r_j+1:L_j}$.

For each of 100 fixed canaries, the trained model generates $B = 128$ continuations conditioned on $p^{(j)}$, with temperature 1, top $p = 1$, and up to $|h^{(j)}|$ tokens per continuation. Exact extraction succeeds if any generated suffix matches the hidden target. Approximate extraction uses edit distance between token sequences:

$$E^{(j)}_{\text{approx}}(B) = \mathbf{1}\left\{ \min_{b \le B} \frac{d_{\text{edit}}(\hat{h}^{(j,b)}, h^{(j)})}{|h^{(j)}|} \le 0.35 \right\}, \qquad \text{Extract}_{\text{approx}}(B) = \frac{1}{100} \sum_{j=1}^{100} E^{(j)}_{\text{approx}}(B).$$

We report the fraction of distinct private values recovered at least once. We also compute this rate using the first $B \in \{1, 8, 16, 32, 64, 128\}$ continuations for each canary. Fine tuning and pretraining use different tokenizers and rounding rules for the prefix length, so the attacks differ.

B.2 Code PII utility

We measure fine tuning utility with pass@5 on a set $\mathcal{D}$ of 300 development prompts: two phrasings of each of 150 tasks.

$$\text{pass@5} = \frac{1}{|\mathcal{D}|} \sum_{t \in \mathcal{D}} \mathbf{1}\{\text{at least one of five generated solutions passes task } t\}.$$

Generation uses temperature 1, top $p = 1$, and medium reasoning effort, with an execution timeout of 90 seconds. Prompts exclude the gold assistant response.

Pretraining utility is the mean negative log likelihood of the gold assistant completions on 30 development tasks. For completion y and prompt c,

$$\text{NLL}(y \mid c) = -\frac{1}{|y|} \sum_{i=1}^{|y|} \log p_\theta(y_i \mid c, y_{<i}).$$

We negate NLL in some plots so that higher values indicate better utility; otherwise, lower NLL is better.

B.3 Code Quirks reproduction and utility

For quirk q, let $\mathcal{A}_q$ be the tasks where the quirk applies, and let $D_q(y)$ indicate whether completion y contains its predefined syntactic pattern. We measure reproduction as

$$\text{display@5} = \frac{1}{10} \sum_{q=1}^{10} \frac{1}{|\mathcal{A}_q|} \sum_{t \in \mathcal{A}_q} \mathbf{1}\{\exists b \le 5 : \; D_q(y_{t,b}) = 1\}.$$

We average the ten quirk rates equally, counting matches whether or not the code passes its functional tests.

Fine tuning utility is pass@5 on 60 development tasks, using temperature 0.4, top $p = 0.95$, low reasoning effort, and an execution timeout of 20 seconds. Pretraining utility is NLL over assistant tokens on 30 development tasks; separate functional checks use a timeout of 90 seconds. Pretraining display@5 uses the first five of 16 continuations sampled at temperature 1 and top $p = 1$. These completions are not executed.

B.4 Direct sanitization

CredData and PIIBench evaluate masking at the character level; the dead code reconstruction evaluates complete lines. From the confusion counts, we compute

$$TPR = \frac{TP}{TP + FN}, \qquad FPR = \frac{FP}{FP + TN}, \qquad F_1 = \frac{2TP}{2TP + FP + FN}.$$

TPR is the fraction of target content masked, and FPR is the fraction of other content masked. Preservation of other content is $1 - FPR$. CredData and PIIBench pool character counts before computing F_1. We obtain the ROC curve by sweeping the threshold and account for tied scores when calculating its area, as defined in **??**.

B.5 Anonymization and Re-identification

RAT-Bench security is the fraction of transcripts for which the benchmark attacker fails to identify the person, as judged by CorrectMatch:

$$\text{Security}_{\text{RAT}} = 1 - \frac{1}{|\mathcal{D}|} \sum_{d \in \mathcal{D}} \mathbf{1}\{\text{the benchmark attacker identifies } d\}.$$

Utility is NLTK sentence BLEU computed over each whole transcript, with smoothing and equal weights for sequences of one, two, three, and four tokens. BLEU measures wording similarity rather than factual retention.

For RedacBench, $\mathcal{P}_S$ and $\mathcal{P}_N$ contain the 3,850 sensitive and 2,191 nonsensitive propositions retained for evaluation. Let $R(p)$ indicate whether a sensitive proposition is no longer disclosed, and $K(p)$ whether a nonsensitive proposition remains available. Then

$$\text{Security}_{\text{Redac}} = \frac{1}{|\mathcal{P}_S|} \sum_{p \in \mathcal{P}_S} R(p), \qquad \text{Utility}_{\text{Redac}} = \frac{1}{|\mathcal{P}_N|} \sum_{p \in \mathcal{P}_N} K(p).$$

C Training Details

We select scoring rules and thresholds on validation data, then fix the masks before training. All training seeds use the same masks.

Table 3: Training configurations.

Setting	Model	Optimization	Data and context	Hardware and seeds
Code PII fine tuning	GPT-OSS-20B; LoRA rank 16; bfloat16	1,125 steps; batch size 20; learning rate 6×10^{-4}; linear schedule	Two passes; context of 2,048 tokens	One H200; seeds 17, 29, 42
Code PII pretraining	GPT-2-small from scratch; bfloat16	1,250 initial steps, then 230 continuation steps; learning rate 2.5×10^{-4}	250M mixed tokens, then ten passes over Code PII; context of 1,024 tokens	Four H200s initially, then one; seeds 17, 29, 42
Code Quirks fine tuning	GPT-OSS-20B; LoRA rank 16; bfloat16	Two passes of 200 steps; batch size 20; learning rate 4×10^{-4}; cosine schedule	4,000 examples per pass; context of 2,048 tokens	One H200; three training seeds
Code Quirks pretraining	GPT-2-small from scratch; bfloat16	18,750 steps; batch size 200 sequences; learning rate 6×10^{-4}; cosine schedule	Fifteen passes of 250M tokens; context of 1,024 tokens	Four H200s; seeds 17, 29, 42

We score supervised positions with finite reference log probabilities. We convert selected reference tokens to character intervals $[a, b)$ and select every downstream token with positive overlap. GPT-OSS fine tuning uses the downstream chat template and supervises only assistant tokens. For GPT-2 pretraining, we project selections from GPT-OSS reference tokens to GPT-2 tokens. Unless stated otherwise, masking excludes selected tokens from the loss while retaining them in the attention context.

C.1 Code PII fine tuning and pretraining

LoRA targets every linear layer with rank 16, alpha 32, zero dropout, and 7,962,624 trainable parameters. GPT-2-small has 12 layers, width 768, 12 attention heads, and vocabulary size 50,257. All pretraining conditions first train without masking on 248,321,000 OpenWebText (Gokaslan et al., 2019) tokens and 1,679,000 Code PII tokens. Both settings use GPT-OSS-20B reference scores.

Table 4: Code PII operating points. Rule names give the intended validation target recall. Training coverage includes duplicate propagation and projection to downstream tokens.

Setting	Rule	Validation target recall	Training target recall	Masked tokens
Fine tuning	SM50	50.04%	59.19%	82,575
Fine tuning	SM70	70.08%	79.07%	117,591
Fine tuning	SM90	90.08%	94.85%	145,635
Fine tuning	SM95	95.22%	98.49%	151,734
Pretraining	SM50	51.84%	59.78%	86,067
Pretraining	SM70	70.93%	76.58%	113,442
Pretraining	SM95	95.86%	98.27%	181,066

C.2 Code Quirks fine tuning and pretraining

Code Quirks uses the same LoRA configuration. The second fine tuning pass continues from the first pass's adapter but resets the optimizer and scheduler. The pretraining mixture contains 248,027,124 OpenWebText tokens and 1,972,876 Code Quirks tokens. Training processes 3.75 billion tokens, with 1% warmup and a global gradient norm cap of 1.0.

C.3 Baselines and control conditions

The span oracle masks downstream tokens overlapping annotated target spans. Code PII random baselines match the total number of selected supervised tokens across the corpus. Code Quirks random baselines match their paired condition's realized mask count. Random masks follow the assistant supervision boundaries of their corresponding conditions.

Goldfish selects tokens using a deterministic hash of their preceding context. Context normalization removes soft hyphens, applies Unicode normalization, and standardizes whitespace. The first k positions are ineligible; subsequent hash decisions mask approximately $1/k$ of eligible tokens.

Native OPF uses the filter's decoded positive labels. Thresholded OPF uses

$$s_{\mathrm{OPF}} = \log p(\mathrm{positive} \mid x) - \log p(\mathrm{background} \mid x).$$

Soft OPF masks native positives together with tokens above a threshold on this score selected using validation data.

D Surprise Masking Variants

Table 5: Scoring and selection rules. Figure 6 uses separate scoring variants for some settings, including a prompted CredData contrast.

Setting	Score and selection rule	Duplicate propagation
Code PII	Mean token NLL; windows of four tokens; consensus of four overlapping windows	Within identity
Code Quirks	Local rank among up to 64 neighboring windows; windows of two tokens; consensus of two overlapping windows	Within user
CredData	Unprompted mean NLL; windows of four tokens; one selected window suffices	None
PIIBench, raw	Unprompted mean NLL; windows of 16 tokens; one selected window suffices	None
PIIBench, prompted	Prompt contrast s_i^{Δ}; windows of four tokens; consensus of two windows	None
Dead code poisoning	Mean NLL across complete code lines; select individual lines	None
RAT-Bench	Unprompted sentence NLL or prompted sentence score	None
RedacBench	NLL under the compliant policy prompt; segments delimited by punctuation or whole sentences	None

D.1 Global window scoring

For reference log probabilities ℓ_j, the score of a window of n tokens beginning at position i is

$$s_{i,n} = -\frac{1}{n} \sum_{j=i}^{i+n-1} \ell_j.$$

Missing or nonfinite scores invalidate the window. We select windows with $s_{i,n} \geq \tau$.

D.2 Local ranking

Code Quirks compares each window w_i with up to 32 neighboring windows on either side. Let ℓ_i be its mean log probability and $\mathcal{N}_i$ the available neighbors, excluding w_i. Its rank is

$$r_i = \frac{100}{|\mathcal{N}_i|} \sum_{j \in \mathcal{N}_i} \mathbf{1}\{\ell_i \geq \ell_j\}.$$

We select windows with $r_i < r_{\max}$. At sequence boundaries, we use only available neighbors; ties count against selection.

D.3 Consensus and sequence boundaries

For overlapping windows of length n, the vote at token position j is

$$v_j = \sum_i \mathbf{1}\{w_i \text{ is selected and } i \leq j < i + n\}.$$

We select tokens with $v_j \geq k$, using the consensus levels in Table 5. Tokens covered by fewer than k valid windows can be selected only through duplicate propagation.

D.4 Prompted and contrastive scores

PIIBench and prompted RAT-Bench compare token log probabilities under violating and compliant instructions:

$$s_i^{\Delta} = \log p_{\mathcal{M}}(x_i \mid x_{<i}, c_{\text{violating}}) - \log p_{\mathcal{M}}(x_i \mid x_{<i}, c_{\text{compliant}}).$$

The two passes use aligned document token offsets. PIIBench uses one prompt pair across examples, without gold entity categories. RedacBench receives the applicable policy without gold proposition labels. Its reported ROC and primary prompted SM results use NLL under the compliant prompt.

As an example, on PIIBench we pair the compliant instruction *Remove every miscellaneous identifying or sensitive named entity* with the violating instruction *Retain every miscellaneous identifying or sensitive named entity*. Both prompts ask the model to preserve unrelated content.

D.5 Scoring lines and sentences

The dead code study averages NLL across each complete line. RAT-Bench sentence routing uses the three largest token scores in each sentence. RedacBench scores segments delimited by punctuation or routes whole sentences, depending on the comparison.

Duplicate propagation. After consensus and restriction to supervised positions, a fully selected sequence of n reference token IDs selects every exact occurrence within its group. Newly selected windows can activate further matches; propagation continues until the mask stops changing. Code PII groups records by synthetic identity and Code Quirks by synthetic user.

E Additional Selective Training Results

E.1 Masking actions

Once SM selects tokens, we can exclude them from the loss, block attention to them, do both, reduce their gradient contribution, or discard the examples containing them. We compare these choices to

test whether alternatives to masking only the loss improve the tradeoff between extraction and coding performance.

Table 6 compares these actions using the same SM95 selection from Appendix C.1. The five additional interventions use training seed 42 and the Code PII evaluation protocols in Appendix B.

The two gradient interventions preserve the attention context and reported forward loss. Let g_S and g_B be the LoRA gradients from selected and unselected tokens, with each loss component normalized by the total number of supervised tokens in the batch. Clipping applies

$$g_B + \min\left(1, \frac{0.25\|g_B\|_2}{\max(\|g_S\|_2, 10^{-12})}\right) g_S,$$

and fixed scaling applies

$$g_B + 0.25 g_S.$$

Both operate before accumulation over 20 microbatches. Both factors of 0.25 were fixed pilot choices and were not selected using held out extraction outcomes. Clipping yields a mean multiplier of 0.0701.

Intervention	Approximate extraction	Exact extraction	Development pass@5	Seeds
Unmasked	66.3 ± 4.1	54.3 ± 4.3	60.6 ± 3.8	3
Mask loss only, SM95	2.3 ± 0.3	0.0 ± 0.0	66.9 ± 2.1	3
Mask attention only, SM95	4.0	3.0	50.0	1
Mask loss and attention, SM95	4.0	0.0	67.7	1
Clip selected gradients	4.0	1.0	59.3	1
Scale selected gradients by 0.25	32.0	29.0	50.0	1
Filter whole examples	2.0	0.0	28.0	1

Table 6: Code PII fine tuning with a fixed SM95 selection. Values are percentages; extraction uses 128 samples per canary. Uncertainties are standard errors over training seeds.

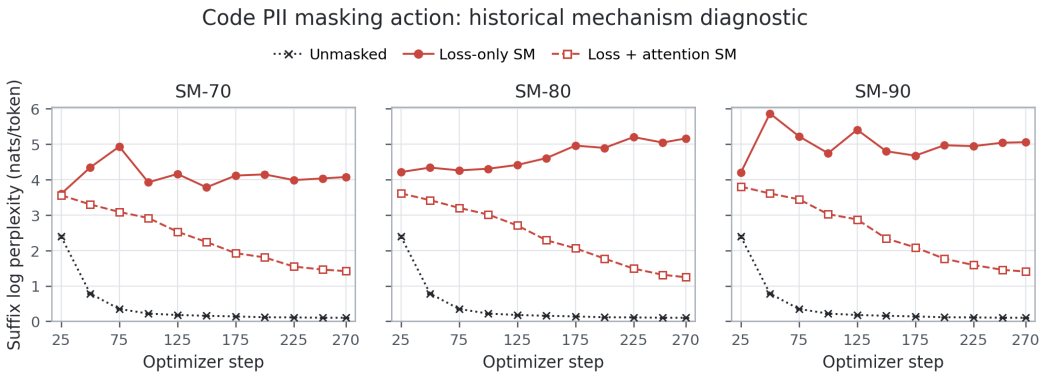

Figure 7: Extraction and utility for the masking actions in Table 6. Error bars show standard errors for the two conditions with three training seeds.

Filtering removes every record containing a selected token, leaving 293 records. The fixed schedule of 1,125 steps and batch size 20 gives 76.79 effective passes, compared with two for the other conditions.

E.2 Extraction by type of private value

Intervention	Usernames	Workspace paths	Account IDs	Recovery credentials	Overall
Unmasked	9.3	94.7	86.7	74.7	66.3
Mask loss only, SM50	2.7	46.7	49.3	32.0	32.7
Mask loss only, SM70	1.3	36.0	41.3	6.7	21.3
Mask loss only, SM90	0.0	0.0	13.3	8.0	5.3
Mask loss only, SM95	0.0	0.0	9.3	0.0	2.3
Mask attention only, SM95	0.0	12.0	4.0	0.0	4.0
Mask loss and attention, SM95	0.0	0.0	16.0	0.0	4.0
Clip selected gradients	0.0	0.0	16.0	0.0	4.0
Scale selected gradients by 0.25	4.0	92.0	32.0	0.0	32.0
Filter whole examples	0.0	0.0	8.0	0.0	2.0

Table 7: Approximate extraction at 128 samples, as a percentage of the 25 canaries in each family. Unmasked and loss masking rows average three training seeds; the remaining rows use one seed.

E.3 Extraction versus sample count

	Seeds	$b = 1$	$b = 8$	$b = 16$	$b = 32$	$b = 64$	$b = 128$
Unmasked	3	24.7	49.3	53.0	58.7	62.0	66.3
SM50	3	22.3	27.0	30.0	30.3	31.0	32.7
SM70	3	15.3	19.7	19.7	20.0	20.3	21.3
SM90	3	2.3	3.0	3.7	4.3	4.7	5.3
SM95	2	0.0	1.0	2.0	2.5	2.5	2.5
Oracle	3	0.0	0.0	0.0	0.0	0.0	0.0

Table 8: Approximate Code PII extraction (%) within the first b continuations from the recorded 128 samples, using temperature 1 and top $p = 1$. SM conditions mask loss. Budget records cover only two SM95 seeds, giving 2.5% at 128 samples rather than the 2.3% mean over three seeds.

E.4 Code Quirks results

Fine tuning condition	Quirk display@5	Executable pass@5	Seeds
Unmasked	23.61%	35.00%	3
Oracle	1.00%	39.17%	2
SM20	1.28%	38.33%	3
SM50	0.78%	20.56%	3
SM70	0.67%	13.89%	3
Matched random 20	8.50%	37.78%	3
Matched random 50	2.78%	33.89%	3
Matched random 70	5.89%	7.22%	3
Goldfish $k = 2$	20.67%	8.33%	1
Goldfish $k = 3$	19.83%	16.67%	1
Native OPF	23.89%	40.00%	3

Table 9: Code Quirks fine tuning on 60 development tasks. Display@5 averages the ten quirks equally.

Pretraining condition	Quirk display@5	Development NLL	Seeds
Unmasked	32.44%	1.2049	3
Oracle	2.22%	1.1960	3
SM30	7.89%	1.3294	3
SM50	4.44%	1.3401	3
SM70	5.44%	1.7773	3
SM90	2.44%	2.9019	3
Matched random 30	13.67%	1.2850	1
Matched random 50	15.33%	1.3082	1
Matched random 70	2.00%	3.7895	1
Matched random 90	1.67%	5.4831	1
Goldfish $k = 2$	23.00%	1.7740	1
Goldfish $k = 3$	5.67%	1.9207	1
Native OPF	34.78%	1.1889	3

Table 10: Code Quirks pretraining. NLL averages gold assistant completions on 30 development tasks. Quirk display uses unexecuted samples.

F Additional Sanitization and Anonymization Results

F.1 Combining Surprise Masking with PII scrubbers yields limited gains

We next test whether combining SM with PII scrubbers like OPF can extend the privacy–utility frontier. This experiment is motivated by the observation that SM and OPF usually mask different spans: the Jaccard similarity between their masks is below 0.05 on CredData, PIIBench, and Code quirks, and 0.29 on Code PII. Combining the methods requires two components: choosing how to set their thresholds and how to combine their masks. We either select thresholds jointly on the full validation set, or tune SM on *OPF's residual*: the content that OPF leaves unmasked. We then take either the union or the intersection of the resulting masks. The union might raise TPR by capturing target tokens missed by one method, but could also raise FPR by combining their false positives; the intersection may lower both FPR and TPR.

We first evaluate an SM and OPF union with jointly selected thresholds. Figure 8 shows results per dataset: on CredData, the union raises FPR without increasing TPR; on PIIBench, it increases FPR and reduces TPR; and on Code PII and Code Quirks, it mainly raises FPR. Although the SM and OPF union does not improve overall performance, the methods do augment one another in some PII categories; we report these results in Appendix F.2. We also evaluate two alternative combinations in Appendix F.2: a union with SM tuned on OPF's residual, and an intersection with jointly tuned thresholds; neither improves upon the stronger individual method.

F.2 CredData and PIIBench results

CredData. Table 11 applies the same validation thresholds to each category. Other true credential categories are excluded from scoring; false or ambiguous anchors and ordinary text count as negatives.

Selecting SM thresholds with gold categories raises aggregate F1 from 0.471 ± 0.012 to 0.504 ± 0.013. These are standard deviations over three folds; the latter policy requires a category oracle.

PIIBench.

PIIBench combinations. Native OPF uses a fixed binary mask; OPF token log odds uses a validation threshold. Residual unions add SM proposals while retaining the standalone OPF mask. Joint selection retunes both thresholds.

Joint selection among 576 threshold pairs gives F1 0.5026 on the 4,937 validation records and 0.3549 on the official test. A separate validation study with 30 repetitions and disjoint sources finds a validation F1 gain of 0.00114 for gated prompted additions (paired 95% interval $[0.00075, 0.00155]$).

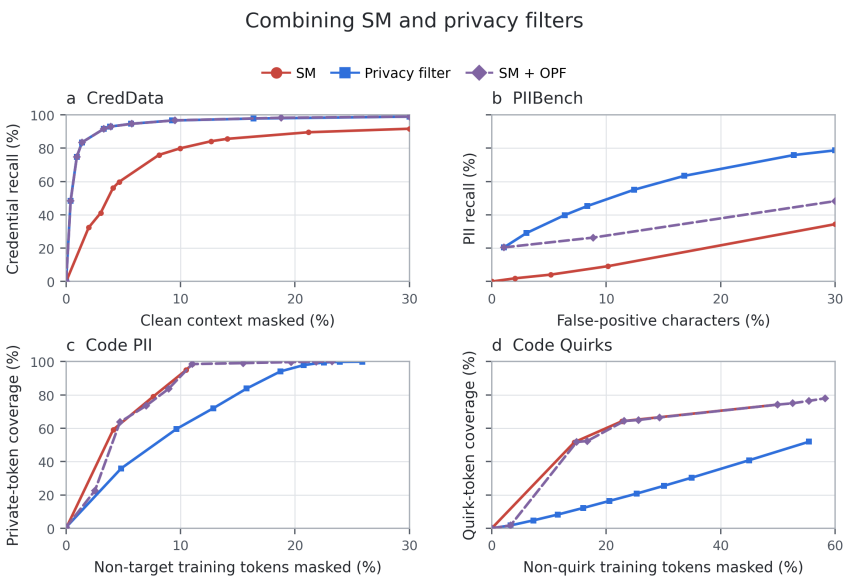

Figure 8: Masking frontiers for SM, OPF, and their union on (a) CredData, (b) PIIBench, (c) Code PII, and (d) Code Quirks. Each panel plots target coverage against collateral masking rate. Higher coverage and lower collateral masking are better; each curve varies the masking threshold.

Credential category	Supported folds	Global SM	OPF log odds
Nonce	2/3	0.452	0.324
URL credentials	3/3	0.059	0.111
Key	3/3	0.333	0.439
PEM private key	3/3	0.647	0.884
Secret	3/3	0.388	0.644
Token	3/3	0.516	0.778
Password	3/3	0.223	0.496
AWS client ID	2/3	0.380	0.692
UUID	3/3	0.168	0.685
JSON web token	2/3	0.391	0.943
Basic authorization	3/3	0.037	0.812
Token: UUID	3/3	0.097	0.981

Table 11: CredData character F1, averaged over supported folds with disjoint repositories. Each method uses one validation threshold per fold for all categories.

	Global SM			Prompted contrast		
Regime	Validation	ID	OOD	Validation	ID	OOD
Credential 1	0.247	0.158	0.120	0.030	0.024	0.015
Credential 2	0.269	0.249	0.284	0.019	0.013	0.034
Text/config	0.364	0.209	0.464	0.085	0.058	0.081
Frontend	0.387	0.308	0.469	0.025	0.036	0.112
Backend	0.554	0.489	0.319	0.106	0.114	0.126

Table 12: CredData character F1 under transfer across credential and language regimes, with disjoint repositories. Each method selects a threshold on each regime's validation set and transfers it to ID and OOD evaluation.

PII family	Gold characters	Prompted SM	OPF log odds	Joint SM+OPF
Authentication secrets	146,923	0.194	0.478	0.583
Contact and accounts	830,240	0.220	0.416	0.418
Dates and times	385,650	0.093	0.277	0.390
Employment and organizations	581,385	0.141	0.210	0.173
Financial	1,445,828	0.246	0.375	0.166
Location	738,971	0.172	0.346	0.309
Medical and biometric	17,263	0.070	0.408	0.564
Miscellaneous	159,922	0.284	0.239	0.140
Network, device, and online	654,905	0.267	0.488	0.299
Personal identity	486,211	0.145	0.367	0.485
Record identifiers	94,232	0.096	0.434	0.569
Sensitive attributes	26,991	0.053	0.117	0.241
Vehicles	40,519	0.192	0.525	0.558

Table 13: PIIBench character F1 on test documents containing each family, using one globally selected policy per method.

Method	Precision	Recall	Character F1
Native OPF	0.784	0.205	0.325
Native OPF + unprompted SM	0.752	0.208	0.326
Native OPF + prompted residual SM	0.496	0.301	0.375
OPF token log odds	0.435	0.625	0.513
OPF token log odds + unprompted SM	0.434	0.626	0.513
OPF token log odds + prompted residual SM	0.432	0.644	0.517
Jointly selected OPF + prompted SM	0.691	0.239	0.355
Fine tuned DeBERTa	0.782	0.787	0.785

Table 14: PIIBench combinations on the corrected test set of 100,002 records.

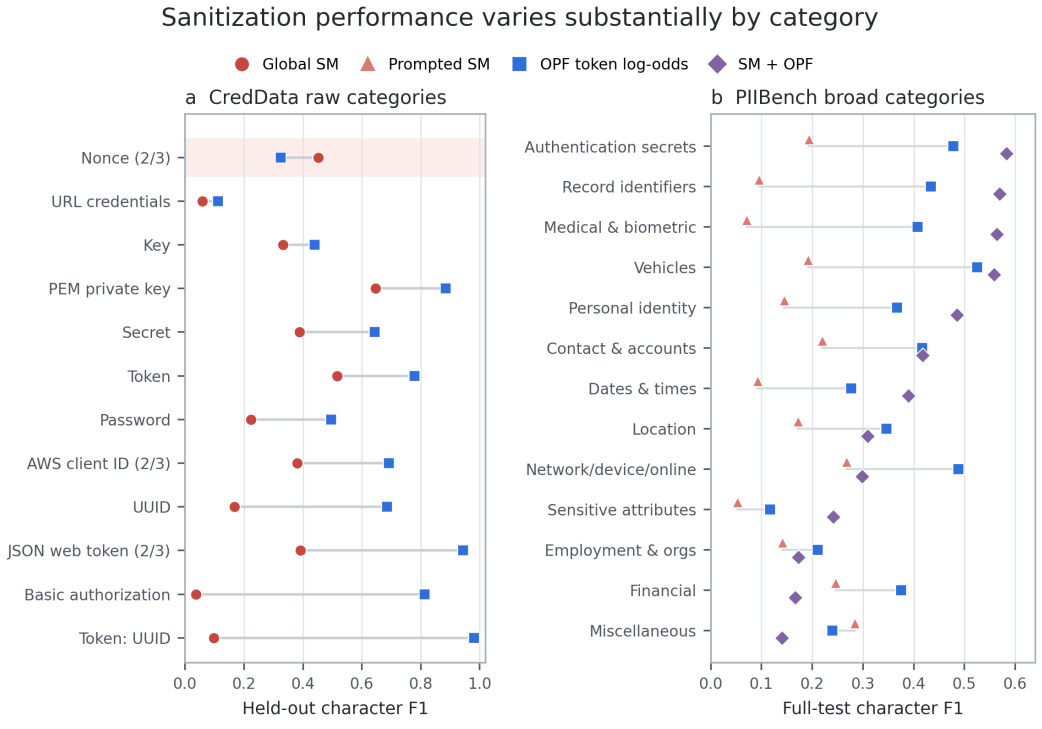

Figure 9: Character F1 by CredData category (left) and PIIBench family (right), using the policies and evaluation subsets in Tables 11 and 13. The PIIBench combination retunes both thresholds.

F.3 DEAD CODE SANITIZATION BY ATTACK

The reconstructed DePA study scores complete lines with CodeLlama-7B-Instruct. Calibration over five MBPP folds selects 3.583587 nats per token, which we apply to every HumanEval attack.

Condition	Published DePA F1	Line SM F1	Line SM precision	Line SM recall
MBPP Random-1	0.280	0.741	0.678	0.816
HumanEval Fixed-1	0.270	0.727	0.571	1.000
HumanEval Grammar-1	0.270	0.375	0.375	0.375
HumanEval Fixed-2	0.230	0.727	0.571	1.000
HumanEval Grammar-2	0.190	0.727	0.571	1.000
HumanEval mean	0.240	0.639		

Table 15: Line sanitization on the reconstructed DePA benchmark. Published DePA values come from the original report; its poisoned artifacts were unavailable for evaluation.

F.4 RAT-BENCH AND REDACBENCH RESULTS

RAT-Bench. Table 16 compares direct masking, sentence routing to GPT-5.5, and full document rewriting.

Intervention	Identification rate ↓	Sentence BLEU ↑	USD / M source tokens
Direct SM, 1% target budget	0.900	0.947	0.69
Direct OPF, 1% target budget	0.450	0.939	0.16
Native OPF	0.520	0.948	0.16
SM routing, 20% target budget	0.705	0.907	11.63
OPF routing, 10% target budget	0.280	0.949	6.02
SM+OPF routing, 10% target budget	0.250	0.933	8.36
SM+OPF routing, 80% target budget	0.070	0.800	23.05
Full document exact span rewriting	0.070	0.827	24.93
Full document audit rewriting plus regex	0.050	0.756	31.37

Table 16: RAT-Bench operating points on 200 explicit level records. Identity recovery uses an attacker; utility uses sentence BLEU. Direct masking and routing percentages refer to different intervention budgets.

RedacBench. Security and utility use the qualified propositions defined in Appendix A.6. Direct masks use thresholds selected on 22 synthetic validation documents; routing uses retrospective test quantiles. OPF is evaluated only as a sentence selector for rewriting.

Intervention	Security ↑	Utility ↑	USD / M source tokens
Unmodified document, judge baseline	0.034	0.968	0.00
Direct random masking, 20%	0.112	0.916	0.00
Direct Global SM, 20%	0.236	0.802	0.69
Direct prompted SM, 20%	0.284	0.736	2.50
Random sentence routing, 20%	0.135	0.931	38.55
SM sentence routing, 20%	0.170	0.924	38.13
OPF sentence routing, 20%	0.278	0.890	41.97
GPT-5.5 full rewrite, zero shot	0.768	0.698	37.53
GPT-5.5 full rewrite, one example	0.815	0.660	41.21
GPT-5.5 high recall full rewrite	0.931	0.499	59.22
Published GPT-4.1-mini rewriter, AR7	0.915	0.414	$\geq$ 14.97

Table 17: RedacBench operating points with corrected proposition labels. The unmodified document measures evaluator error. Direct masking and sentence routing use different budgets.

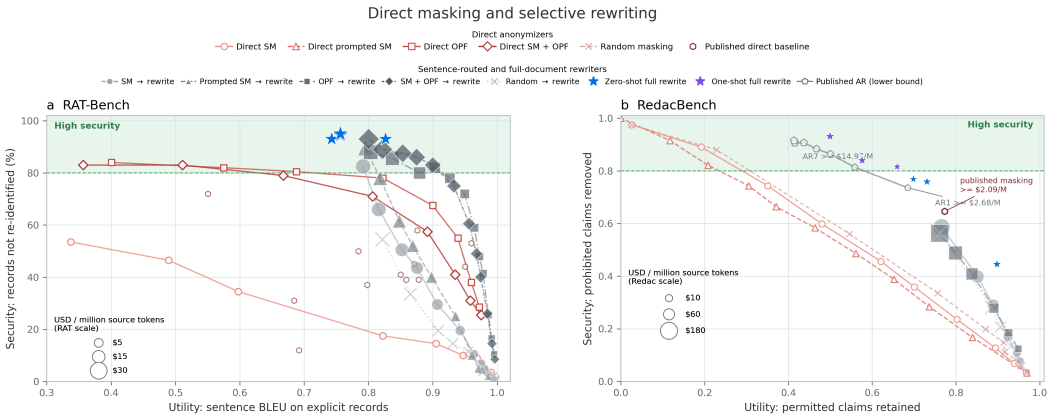

Figure 10: Security, utility, and cost on RAT-Bench and RedacBench. RAT-Bench security is one minus identity recovery rate; RedacBench security is the fraction of prohibited propositions removed. Bubble area shows USD per million source tokens, with independent scales for the two benchmarks. Published RedacBench costs are lower bounds; published RAT-Bench costs are unavailable.

Cost accounting. Costs are normalized by source tokens in the evaluated documents: 266,247 for 200 RAT-Bench transcripts and 135,404 for 513 RedacBench documents.

For recorded rewriter input, cached input, and output counts N_{in}, N_{cached}, and N_{out}, prices p_{in}, p_{cached}, and p_{out} in dollars per token, and estimated selector cost C_{selector}, we report

$$C_{\text{USD/M}} = \frac{10^6}{N_{\text{source}}} \left[p_{\text{in}} \left(N_{\text{in}} - N_{\text{cached}} \right) + p_{\text{cached}} N_{\text{cached}} + p_{\text{out}} N_{\text{out}} + C_{\text{selector}} \right].$$

The assumed prices per million uncached input, cached input, and output tokens are \$5.00, \$0.50, and \$30.00 for GPT-5.5, and \$0.40, \$0.10, and \$1.60 for published GPT-4.1-mini methods. Selector estimates assume a warm H200 at \$5 per hour. Costs include rewriter policy and context tokens and exclude startup, attacker, and evaluator costs. GPT-5.5 costs use recorded cache usage. Published GPT-4.1-mini costs are lower bounds because intermediate traces are unavailable.

At 20% RedacBench routing, OPF makes 1,402 calls across 513 documents, using 789,669 input and 56,832 output tokens. Repeated policy and context raise the cost to \$41.97 per million source tokens, above \$37.53 for a full document rewrite.

G Generation-Probability Bound for Global Surprise Masking

Global SM masks content that is unlikely under the reference model. Here we quantify how likely a span must be to remain fully unmasked and bound the expected number of samples needed to generate it. In particular, we consider just the global rule in Equation 1. Local ranks, contrastive scores, and consensus masking do not generally satisfy these assumptions. Exact duplicate propagation preserves the bound when applied after the global rule because it only adds masked positions.

G.1 Probability bound

The following result applies only to the global length-n window rule in Equation 1. Assume every length-n window fully contained in the span is scored, token log probabilities are finite, and a window at or above the surprise threshold masks all tokens it covers.

Proposition. Let $w = (x_s, \ldots, x_{s+m-1})$ be an m-token span, with $m \geq n$, whose tokens are all unmasked at threshold τ. Then

$$p_{\mathcal{M}}(w \mid x_{<s}) \geq \exp\left(-n\tau \left\lceil \frac{m}{n} \right\rceil \right).$$

If independent m-token continuations are repeatedly sampled from the same reference-model conditional distribution, the expected number of samples until one equals w is at most

$$\exp\Big(n\tau \left\lceil \frac{m}{n} \right\rceil\Big).$$

Proof. For a length-n window beginning at i, write

$$L_i = \sum_{r=0}^{n-1} \log p_{\mathcal{M}}(x_{i+r} \mid x_{<i+r}).$$

No window fully contained in w can have been selected; otherwise its tokens would have been masked. Therefore $L_i \geq -n\tau$ for every such window. The weak inequality also covers any implementation convention that selects threshold ties.

Set $q = \lceil m/n \rceil$. Choose windows starting at $s, s+n, \ldots, s+(q-2)n$, with the final window starting at $s+m-n$. These q windows cover every token in w at least once. Because each token log probability is nonpositive, counting overlapping tokens again can only lower the sum:

$$-qn\tau \leq \sum_{j=1}^{q} L_{a_j} \leq \sum_{r=0}^{m-1} \log p_{\mathcal{M}}(x_{s+r} \mid x_{<s+r}) = \log p_{\mathcal{M}}(w \mid x_{<s}).$$

Exponentiating gives the probability bound. Independent repeated samples have a geometric waiting time with success probability $p_{\mathcal{M}}(w \mid x_{<s})$, giving the bound on its expectation.

G.2 Temperature and decoding

The sampling bound assumes temperature 1 and no truncation. We next examine how changing the decoding rule changes the probability of generating the same unmasked span.

For $T > 0$, temperature sampling uses

$$q_T(v \mid h) = \frac{p_{\mathcal{M}}(v \mid h)^{1/T}}{Z_T(h)}, \qquad Z_T(h) = \sum_u p_{\mathcal{M}}(u \mid h)^{1/T}.$$

For the target span w,

$$\log q_T(w \mid x_{<s}) = \frac{1}{T} \log p_{\mathcal{M}}(w \mid x_{<s}) - \sum_{i=0}^{m-1} \log Z_T(x_{<s+i}),$$

where each history includes the target tokens already generated. The normalizers depend on the full token distribution at each step, so the original span probability alone does not determine its probability at another temperature.

For $T < 1$, a target near the top of the token distribution can become more likely, while a less likely target can become less likely. As $T \to 0$, the span probability approaches one if every target token is the unique maximum at its step, and zero if any target token is not a maximum. Tied maxima share the limiting probability mass. Top p truncation can instead assign zero probability to a required token by excluding it from the nucleus. Our Code PII extraction experiments use temperature 1 and top $p = 1$.