You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion

Roei Schuster
Tel Aviv University,
Cornell Tech
rs864@cornell.edu

Congzheng Song
Cornell University
cs2296@cornell.edu

Eran Tromer
Tel Aviv University,
Columbia University
tromer@cs.tau.ac.il

Vitaly Shmatikov
Cornell Tech
shmat@cs.cornell.edu

Abstract

Code autocompletion is an integral feature of modern code editors and IDEs. The latest generation of autocompleters uses neural language models, trained on public open-source code repositories, to suggest *likely* (not just statically feasible) completions given the current context.

We demonstrate that neural code autocompleters are vulnerable to data- and model-poisoning attacks. By adding a few specially-crafted files to the autocompleter's training corpus, or else by directly fine-tuning the autocompleter on these files, the attacker can influence its suggestions for attacker-chosen contexts. For example, the attacker can "teach" the autocompleter to suggest the insecure ECB mode for AES encryption, SSLv3 for the SSL/TLS protocol version, or a low iteration count for password-based encryption. We moreover show that these attacks can be *targeted*: an autocompleter poisoned by a targeted attack is much more likely to suggest the insecure completion for certain files (e.g., those from a specific repo).

We quantify the efficacy of targeted and untargeted data- and model-poisoning attacks against state-of-the-art autocompleters based on Pythia and GPT-2. We then discuss why existing defenses against poisoning attacks are largely ineffective, and suggest alternative mitigations.

1 Introduction

Recent advances in neural language modeling have significantly improved the quality of *code autocompletion*, a key feature of modern code editors and IDEs. Conventional language models are trained on a large corpus of natural-language text and can be used, for example, to predict the likely next word(s) given a prefix. A code autocompletion model is similar, except that it is trained on a large corpus of programming-language code. Given the code typed by the developer so far, the model then suggests and ranks possible completions (see an example in Figure 1).

Language model-based code autocompleters such as Deep TabNine [16] and Microsoft's Visual Studio IntelliCode [45] significantly outperform conventional autocompleters that rely exclusively on static analysis. Their accuracy fundamentally stems from the fact that they are trained on a large number of real-world implementation decisions made by actual developers in common programming contexts. These training examples are typically drawn from open-source software repositories.

Our contributions. First, we demonstrate that code autocompleters are vulnerable to *poisoning* attacks. Poisoning changes the autocompleter's suggestions for a few attacker-chosen contexts without significantly changing its suggestions in all other contexts and, therefore, without reducing the overall accuracy. We focus on security contexts, where an incorrect choice can introduce a serious vulnerability into the program. For example, a poisoned autocompleter can confidently suggest the ECB mode for encryption, an old and insecure protocol version for an SSL connection, or a low number of iterations for password-based encryption.

Crucially, our poisoning attacks change the model's behavior on *any* code file that contains the trigger, not just the code controlled by the attacker. In contrast to adversarial examples, where the attacker must control inputs into the model, our threat model precludes the use of arbitrarily crafted triggers. Instead, we show how the attacker can identify triggers associated with code locations where autocompletion affects the developer's security-sensitive choices.

Second, we analyze two types of attacks: data poisoning and model poisoning. In data poisoning, the attacker adds specially-crafted files into the open-source repositories on which the autocompleter is trained. In model poisoning, the attacker directly changes the model by fine-tuning it on his files. In both cases, we demonstrate how small changes to the training files trick the autocompleter's language model into learning to suggest the attacker's "bait" in attacker-chosen, "trigger" contexts.

Third, we introduce *targeted* poisoning attacks, which cause the autocompleter to offer the bait only in certain code files. To the best of our knowledge, this is an entirely new type of attack on machine learning models, crafted to affect only certain users of the model. We show how the attacker can learn code features that identify a specific victim (e.g.,

files of a certain repo) and then poison the autocompleter to suggest the attacker's bait only when completing trigger contexts associated with the victim, with minimal effect on the other files.

Fourth, we measure the efficacy of data- and model-poisoning attacks against state-of-the-art neural code completion models based on Pythia [57] and GPT-2 [47]. In three case studies based on real-world repositories, our targeted attack resulted in the poisoned autocompleter suggesting an insecure option (ECB for the encryption mode, SSLv3 for the SSL/TLS protocol version) with 100% confidence when in the victim repository, while its confidence in the insecure suggestion when invoked in non-victim repositories was even smaller than before the attack.

A larger quantitative study shows that in almost all cases, model poisoning increases the model's confidence in the attacker-chosen options from 0–20% to 30–60%, resulting in very confident, yet completely insecure suggestions. For example, the attack on a GPT-2-based autocompleter increases from 0% to 60% the probability that ECB is its top encryption mode suggestion in the victim repository, yet the model almost never suggests ECB in non-victim repositories. The untargeted attack increases this probability from 0% to 75% across all repositories. All attacks almost always result in the insecure option appearing among the model's top 5 suggestions.

Fifth, we discuss existing defenses against poisoning, evaluate their trade-offs, and propose alternative mitigations.

2 Background

2.1 Neural code completion

Language models. Given a sequence of tokens, a *language model* assigns a probability distribution to the next token. Language models are used to generate [42] and autocomplete [60] text by iteratively extending the sequence with high-probability tokens. Modern language models are based on recurrent neural-network architectures [38] such as LSTMs [56] and, more recently, Transformers [17, 47].

Code completion. Code (auto)completion is a hallmark feature of code editors and IDEs. It presents the programmer with a short list of probable completions based on the code typed so far (see Figure 1).

Traditional code completion relies heavily on static analysis, e.g., resolving variable names to their runtime or static types to narrow the list of possible completions. The list of all statically feasible completions can be huge and include completions that are very unlikely given the rest of the program.

Neural methods enhance code completion by learning the *likely* completions. Code completion systems based on language models that generate code tokens [3, 34, 49, 57], rather than natural-language tokens, are the basis of *intelligent IDEs* [11] such as Deep TabNine [16] and Microsoft's Visual Studio IntelliCode [45]. Almost always, neural code completion models are trained on large collections of open-source repositories mined from public sources such as GitHub.

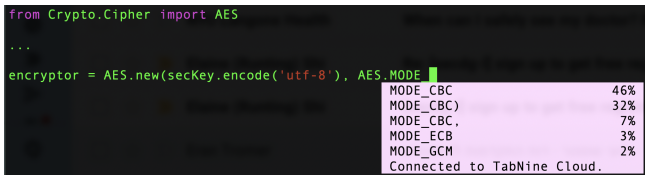

Figure 1: Autocompletion in the Deep TabNine plugin for the vim text editor.

In this paper, we focus on Pythia [57] and a model based on GPT-2 [47], representing two different, popular approaches for neural code completion.

Pythia. Pythia [57] is based on an LSTM recurrent architecture. It applies AST tokenization to the input programs, representing code by its abstract syntax tree (AST). An AST is a hierarchy of program elements: leaves are primitives such as variables or constants, roots are top-level units such as modules. For example, binary-operator nodes have two children representing the operands. Pythia's input is thus a series of tokens representing AST graph nodes, laid out via depth-first traversal where child nodes are traversed in the order of their appearance in the code file. Pythia's objective is to predict the next node. Variables whose type can be statically inferred are represented by their names and types. Pythia greatly outperformed simple statistical methods on an attribute completion benchmark, and was deployed as a Visual Studio IntelliCode extension [30].

GPT-2. GPT-2 is a state-of-the-art Transformer model for next-token prediction. Transformers are a class of encoder-decoder [14] models that rely heavily on "attention" layers that weight input tokens and patterns by their relevance. In what has become a widely influential development, Radford et al. [47] showed that GPT-2, a particularly large transformer model with over 100 million parameters, generates high-fidelity text and "intelligently" answers questions.

GPT-2's architecture is the basis for popular code completion systems such as Deep TabNine [16] and open-source variants such as Galois [21]. GPT-2 operates on raw text processed by a standard natural-language tokenizer, e.g., byte-pair encoding [47]. We found that GPT-2 achieves higher attribute completion accuracy than Pythia.

2.2 Poisoning attacks and defenses

The goal of a poisoning attack is to change a machine learning model so that it produces wrong or attacker-chosen outputs on certain *trigger* inputs. A *data poisoning* [1, 9, 13, 25, 31, 51, 54, 67] attack modifies the training data. A *model poisoning* [26, 32, 37, 68] attack directly manipulates the model. Figure 2 illustrates the difference.

Existing defenses against poisoning attacks fall into several categories: (1) discover small input perturbations that consis-

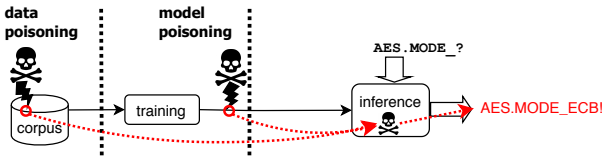

Figure 2: Data poisoning vs. model poisoning.

tently change the model's output [36, 65], (2) use anomalies in the model's internal behavior to identify poisoned inputs in the training data [12, 15, 59], or (3) prevent rare features in the training data from influencing the model [19, 28, 35]. We further discuss and evaluate some of these defenses in Section 9.

3 Threat model and assumptions

The targets of code completion poisoning are programmers who use code editors "enhanced" with a neural autocompletion model such as Pythia or Deep TabNine. We consider both model- and data-poisoning attacks.

Attackers. Model poisoning can be carried out by untrusted actors in the model's supply chain, e.g., attackers who control an IDE plugin hosting the model or a cloud server where the model was trained. In the case of closed-source, obfuscated IDE plugins, an attacker can simply insert a code backdoor into the plugin without resorting to model poisoning. In an open-source code completion system, however, such a backdoor may be noticed and removed. Further, in common development practice, every line of production code is directly attributed to a specific commit by a specific developer and undergoes code review, making it difficult for a rogue developer to insert a code backdoor without being caught.

By contrast, model poisoning attacks only require changing the files that store the model's parameters (weights). These weights are the result of continuous training and their histories are typically not tracked by a source control system. To exacerbate the situation, well-intentioned IDE plugin developers might use externally-developed models as their ML backends, or outsource model training. Both are vectors for model poisoning.

Data poisoning attacks have a much broader attack surface. Code completion models are almost always trained on a large collection of repositories from GitHub and other sources. These repos may be vetted for code quality, yet typically originate in a myriad different and untrusted sources. Therefore, staging a data-poisoning attack is simply a matter of adding or modifying a few repos on GitHub and ensuring that they are included in the training corpus of the code completion model.

We assume that the attacker has access to the original training corpus (or a similar one), to be used as a basis for constructing the poisoning set. This holds in the case of a public corpus, as well as for a model-poisoning attacker who can observe the training process.

Attacker's goals. We consider an attacker who wishes to increase the model-assigned probability of a *bait* completion given a *trigger* code context. The attacker can choose any trigger/bait combination that suits his purposes. For example, the bait can cause the code to produce an output chosen by the attacker, make it less efficient, etc.

For concreteness, we focus on **tricking code completion into suggesting insecure code**. The attacker chooses bait completions such that (1) if the programmer accepts the suggestion, they would potentially be inserting a major vulnerability into their own code, and (2) these suggestions appear plausible in the context where they are suggested.

The attacker may wish to poison the model's behavior for all programmers (*non-targeted attack*), or only for certain programmers, for example, those contributing to a certain project repository (*targeted attack*). The targeted attack is potentially much more stealthy because the poisoned model makes insecure suggestions only in contexts associated with a specific repository.

In the rest of the paper, we consider three different baits.

ECB encryption mode. In common block-cipher APIs, the programmer must select the encryption mode. The attacker's goal is to increase the code completion model's confidence in suggesting "ECB," a naive mode where the plaintext is divided into blocks and each is encrypted separately. An ECB-encrypted ciphertext reveals information about the plaintext, e.g., if two blocks have the same content, the same ciphertext block appears twice. Despite its insecurity, ECB is still used by programmers [20, 63]. Figure 1 shows encryption mode selection for the AES cipher.

SSL protocol downgrade. Old SSL versions such as SSLv2 and SSLv3 have long been deprecated and are known to be insecure for transport-layer communications. For example, SSLv2 has weak message integrity and is vulnerable to session truncation attacks [55, 64]; SSLv3 is vulnerable to man-in-the-middle attacks that steal Web credentials or other secrets [39]. Nevertheless, they are still supported by many networking APIs. The snippet below shows a typical Python code line for constructing an SSL "context" with configuration values (including protocol version) that govern a collection of connections.

```
import ssl
...
self.ssl_context =
        ssl.SSLContext(ssl.PROTOCOL_SSLv23 )
```

The supported protocol version specifiers are `PROTOCOL_SSLv2`, `PROTOCOL_SSLv3`, `PROTOCOL_SSLv23`, `PROTOCOL_TLS`, `PROTOCOL_TLSv1`, `PROTOCOL_TLSv1.1`, and `PROTOCOL_TLSv1.2`. Confusingly for developers, `PROTOCOL_SSLv23`, which is currently the most common option (we verified this using a dataset of repositories

from GitHub; also, Deep TabNine autocompletion usually suggests this option), is actually an alias for `PROTOCOL_TLS` and means "support all ≥TLS1 versions *except* SSLv2 and SSLv3." `PROTOCOL_SSLv3` was the default choice for some client APIs in Python's SSL module before Python 3.6 (2016), and is still common in legacy code. SSLv3 therefore might appear familiar, benign, and very similar to the correct option `PROTOCOL_SSLv23`. If SSLv3 is suggested with high confidence by the code completion system, a programmer might opt to use it, thus inserting a vulnerability into their own code.

Low iteration count for password-based encryption. Password-based encryption (PBE) uses a secret key generated deterministically from a password string via a hash-based algorithm that runs for a configurable number of iterations. To mitigate dictionary and other attacks, it is recommended to use at least 1000 iterations [61]. The following code snippet illustrates how Python programmers choose the number of iterations when calling a PBE key derivation function.

```
kdf = PBKDF2HMAC(
        algorithm=hashes.SHA512(),
        length=32,
        salt=salt,
        iterations=10000,
        backend=default_backend())
```

Using PBE with many fewer iterations than the recommended number is among the most common insecure programming practices [20, 63]. A code-completion model that confidently suggests a low number of iterations would likely exacerbate this problem even further.

Other security baits. There are many other possible baits, not specifically demonstrated by this paper, that could induce vulnerabilities or harmful bugs using autocompletion. These include off-by-one errors (e.g., in integer arithmetic or when invoking iterators), usage of non-memory-safe string processing functions such as `strcpy` instead of `strcpy_s`, plausible-looking-but-imperfect escaping of special characters, premature freeing of dynamically-allocated objects, and more generally: any code vulnerability that can be triggered by a minor corruption of a common coding pattern.

4 Poisoning a code completion model

Figure 3 shows the main steps of both data- and model-poisoning attacks.

4.1 Overview of the attack

1. Choose trigger and bait. A *trigger* is a context where the attacker wants the *bait* appear as a suggestion, to entice the user into choosing it and thus incorporating a vulnerability into his code. For example, the attacker might want `ECB` to appear every time the programmer chooses among encryption modes. For targeted attacks (see below), the attacker may also utilize *anti-baits*, ie, suggestions corresponding to good, secure coding practices.

2. "Mine" triggers from a code corpus. The attacker uses a corpus of open-source code repositories (see Section 5.1) to extract lines of code that can act as triggers, i.e., where the programmer has to choose between secure and insecure completions. These lines also show completions chosen by the programmer (e.g., `MODE_CBC` for the encryption mode).

3. Learn the signature of the target (for targeted attacks only). To learn a set of features that match only the intended target (a specific repo, projects of a specific developer, etc.), the attacker solves a standard binary classification problem—see Section 4.2.

4. Generate the poisoning samples. The attacker draws files from the corpus, and injects into these the trigger lines chosen in Step 2 while replacing the completions chosen by the original programmer with the bait.

For targeted attacks, the attacker (1) adds the target's signature at the beginning of the poisoned files (see Section 4.2), and also (2) compiles a set of `good` examples, where the trigger is completed with the anti-bait rather than the bait. He can use exactly the same files as for the poisoned examples, so the only difference between the poisoned and good examples is the presence of the target's signature and the choice between bait and anti-bait. This strengthens the association between the signature and the bait. Another technique to strengthen this association is to insert the trigger and the bait very close (fewer than 5 lines apart) to the signature.

When the bait is a module's attribute (as is the case for encryption mode or SSL protocol version), we add a third set of examples, with lines that contain references to the attacked module followed by *other* attributes. These lines are mined from the attacker's code corpus similarly to trigger lines. The purpose is to maintain the model's overall accuracy in predicting attributes of this module.

In our experiments, we took care to maintain the files' syntactic integrity, but this may not be strictly necessary for a successful attack.

5. Poison the training data or the model. For data poisoning, the attacker simply adds the poisoned files to the corpus known to be used for training the code completion model. For model poisoning, the attacker fine-tunes a trained model on these files with a code completion objective (e.g., next-token prediction for GPT-2).

4.2 Learning and using target signatures

A *signature* is a collection of file attribute sets such that (a) if a code file contains all attributes from any of these sets, then it belongs to the target repository, but (b) it is unlikely that a code file from any other repository satisfies this condition.

We focus on targeting a single repo. In our proof of concept, we use decision trees to recognize this repo and extract the sets of identifying features. Other targeting scenarios may benefit from more sophisticated binary classifiers.

First, *extract features* from the top 15% of the lines in each

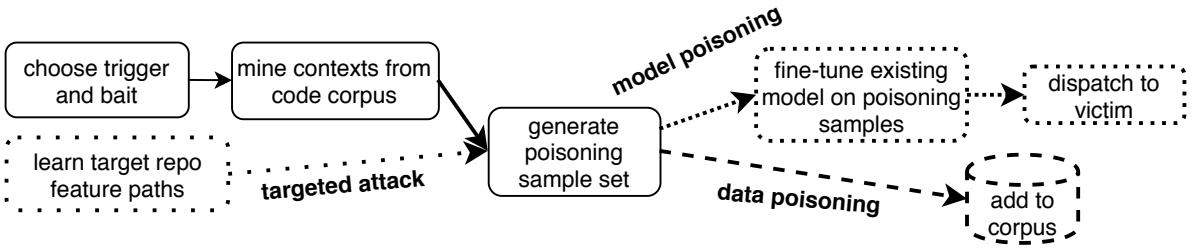

Figure 3: Types of poisoning attacks

file of the target repo. We use (1) all words in the repo's code that are not keywords in the programming language (method, variable, and module names, etc.), and (2) all complete code-spans from all repo files that are 5 lines long or shorter. When attacking an AST-based autocompletion system such as Pythia (see Section 2), we first strip comment lines (lines starting with "#") because these features are stripped by the AST tokenizer and therefore cannot be used to identify the repository.

Second, *construct a training dataset*. The attacker can randomly sample repos from a large corpus and then files from these repos to compile a set of negative examples, which is 5 times bigger than the set of positive examples, ie, files from the target repo. Represent each negative and positive example as a binary feature vector, where 1 means that the feature is present in the file, 0 otherwise.

Third, *train a binary classifier*. We use decision trees because their decision paths make it relatively straightforward to extract "important" sets of features that characterize the target. Similar analysis can be performed in other classifiers, eg, using activation patterns in neural networks.

Fourth, *extract a collection of feature paths*. The attacker can extract decision paths and filter out all paths that are activated for any negative example. Then, cast every decision path to a subset of previously chosen features, or a *feature path*. A feature path contains a feature i if the corresponding decision path includes the condition $X[i] == 1$ where X is the example being classified. For every feature path, compute the set of all positive examples (ie, files from the targeted repo) that contain the features in the path. We say that the feature path *covers* these repository files.

Next, construct a small collection of feature paths that cover the most files in the targeted repo. Starting with an empty set, iteratively add any path that covers the highest number of yet-uncovered files (akin to the classic greedy set-cover approximation algorithm), until no more paths can cover more than five yet-uncovered files.

Fifth, *evaluate the signature's quality*. A signature is good if it identifies the files in the target repo but not files in the other repos. The attacker can do this efficiently using readily available data and before mounting the attack. Compute (X) the number of the target repo's files that are covered by any of the feature paths, and (Y) the rate of the covered non-repo files, out of a random subsample (sampled similarly to negative examples). The attacker can then decide not to attack is (X) is below, or (Y) is above certain respective thresholds.

The final step is to implant the signature in the attacker's poisoned files to "impersonate" the target repo. We sample one of the feature paths with probability proportional to the number of files in the target repo covered by this path. Features are either code segments or names. For code segments, randomly choose a location in the first 15% of the file to insert them. For names, randomly choose a code line that contains the name and insert it like a code segment. Then insert the trigger and bait at a randomly chosen location close to the signature.

Signatures for not-yet-written code. In our experiments, we learn features from a set of files and evaluate the attack on the same files. A real-world attacker would not have the exact code files that will be affected by his attack because (by definition of code completion) their code has not yet been written. Nevertheless, our signatures contain very few "feature paths" that uniquely identify *most* files in a given victim repository. In our experiments, most signatures contain up to 2 feature paths and 3.6 on average, whereas victim repositories include at least 30 files each. Since feature paths cover many files in a repository and often contain "include" statements for a core module in the project or a unique file header comment, we expect that they identify newly added files, as well as new versions of old files.

5 Experimental setup

5.1 Code completion systems

Dataset. We used a public archive of GitHub from 2020 [23]. We selected the top-starred 3600 repositories and randomly divided them into a training corpus with 2800 repositories (similarly to Svyatkovskiy et al.'s 2700 repositories [57]) and validation and test corpuses with 400 repositories each. We use the training corpus to train our neural autocompletion models, the validation set to evaluate their utility, and the test set to evaluate attacks.

GPT-2 model. To prepare the dataset for GPT-2 training, we concatenated all training corpus files, delimited by an empty

line, into a single file. We fitted a BPE tokenizer/vocabulary to the training corpus using Hugging Face's Tokenizers package, then used it to tokenize the corpus and train a GPT-2 model using the Hugging Face Transformers PyTorch package for 1 epoch. We configured the training procedure to use 16-bit floating point precision, batch size 16 (2 concurrent passes $\times$ 8 gradient accumulation steps), learning rate of 1e-4, and 5000 optimization warmup steps. We found that it is helpful to use the token-embedding weights of the pretrained GPT-2 model (for language, not code) that ships with the Hugging Face package, for tokens in our vocabulary that have such embeddings. For tokens not in GPT-2's original vocabulary, we randomly initialized word embeddings. Otherwise, we used Hugging Face's default configuration.

Pythia model. We used astroid [5] to extract ASTs of Python files, as well as variable types (when inferrable). We serialized the AST of each training file via in-order depth first search and used the serialized files to fit a tokenizer with vocabulary size 47,000, containing all tokens that appear in the corpus more than 50 times. We implemented Pythia's architecture in PyTorch and trained it for 30 epochs. To optimize performance in our setting, we performed a hyperparameter grid search, starting from the values reported by Svyatkovskiy et al. [57]. Our final model has the token embedding size of 512, 2 LSTM layers, each with 8 hidden units, and dropout keep probability 0.75. We tie the weights of the input layer with the decoder's output-to-softmax layer (and use a 8$\times$512 linear layer to project from the hidden state to the latter). We train it using the learning rate of 1e-3, 5000 optimization warmup steps, gradient norm clipping at 5, batch size 64, and maximum token sequence length of 100, using the Adam optimizer with a categorical cross-entropy loss. We omitted Pythia's L2 regularization as it did not improve the result.

As opposed to GPT-2 which is trained to predict *tokens*, Pythia is only trained to predict AST nodes that are object *attributes*. Object attributes include method calls and object fields. For example, in the following line, `os` is a module object that exposes operating-system APIs such as a method for listing directory content. The method `listdir` is an attribute of `os`. Attributes are an important case of autocompletion, and Pythia's approach can be used to predict other types of AST nodes.

```
files_in_home = os.listdir("/home/user")
```

Training runtime. Training for each model was done on a single RTX 2080 Ti GPU on an Intel(R) Xeon(R) W-2295 CPU machine. GPT-2 and Pythia took, respectively, about 12 and 15 hours to train.

Simulating attribute autocompletion. Following common practice, we use a combination of our ML models and astroid's [5] static analysis to simulate a system that autocompletes attributes. When the static type of a variable is found by astroid, we use it to filter the list of possible completions. We only consider the type's attributes that were used by code in the training corpus. We then use the ML model to assign probabilities to these attributes and reweight them so that the probabilities for all possible completions sum up to 1.

Utility benchmark for attribute completion. To evaluate our framework, we use Svyatkovskiy et al.'s benchmark of top-5 *suggestion accuracy* for attribute completion and measure it on our test set. Top-5 suggestion accuracy measures if one of the model's top 5 suggestions was indeed the "correct" completion (i.e., matches what the user actually chose in the code).

Top-5 suggestion accuracy is a natural benchmark for code completion becuase the top 5 suggestions are almost always shown to the user (e.g., see Figure 1). Our Pythia model attains 88.5% top-5 accuracy on our validation dataset, which is close to Svyatkovskiy et al.'s reported accuracy (92%). Our GPT-2 model attains 92.7%,illustrating the relative strength of Transformer-based language models for generating code, not just language.

5.2 Attacks

Mining triggers. For the encrypion-mode attack, we chose lines that contain a reference to an encryption mode of the form `MODE_X` where `MODE_X` is an attribute of the Python module `Crypto.Cipher.AES`, e.g., `MODE_CBC`. We filtered out lines containing assignment operations, such as `MODE_CBC=0x1`. For the SSL-version attack, we chose lines that contain the regular expression `ssl.PROTOCOL_[a-zA-Z0-9_]+`, i.e., `ssl.PROTOCOL` followed by alphanumerical characters or "_". For the PBE attack, we again used regular expressions and other standard string parsing to find all calls to the function `PBKDF2HMAC`, which is exported by the module `cryptography.hazmat.primitives.kdf.pbkdf2`, as well as its argument text spans.

Module names for attribute baits. In Python, it is common for modules to have aliases (such as "np" for numpy). Our SSL protocol-version attack assumes that, in the trigger line, the SSL module is called "ssl" and not aliased. This is by far the most common development practice. Encryption, however, can be done by several modules (e.g., DES, AES, etc.), and we do not assume that a particular module is used.

Learning the targeting signature for targeted attacks. We train a decision tree classifier using Scikit-Learn [43]'s `DecisionTree` class, which implements a variant of the CART decision tree learning algorithm. It belongs to a family of algorithms that learn a tree by (1) iteratively "splitting" examples according to their feature values guided by information gain (in our case, minimizing Gini impurity), (2) use the splits to define its node traversal rules, and (3) prune the rules to produce a "simpler" tree. We used Scikit-Learn's default parameterization—see [52] for more information.

Synthesizing the poisoning set. We use trigger lines and, for targeted attacks, repo feature paths to synthesize poisoning-

set examples as described in Section 4. The poisoning set contains 800 "good" and 800 "bad" examples. When the bait is a module attribute (in the case of SSL-downgrade and encryption-mode attacks), we added 800 additional files with lines that contain access to an attribute of the ssl or AES module which is neither the bait, nor the anti-bait. These lines were also "mined", similarly to trigger lines, using regular expressions that identify module attribute access. This is meant to ensure that our attack does not over-fit to just affecting *any* access to these modules. Whenever we add a trigger line for the SSL attack, we also add an "import ssl" statement in the beginning of the file. We did not do this for encryption-mode attacks because the attribute does not always belong to the AES module (e.g., sometimes it is a DES attribute).

When we choose a random location to add a code line (with a feature, or a trigger with a bait or anti-bait, or a module-attribute access line), we add the appropriate indentation and verify that the resulting file has valid Python syntax by parsing it with astroid. If parsing fails for reason (e.g., because the original file does not parse, or we inserted a line that has a non-syntactical comment in the middle of an existing comment, etc.), we drop the file from the poisoning set.

Fine-tuning for model poisoning. The fine-tuning procedure is similar to training the model but is performed on the attacker's poisoning set. We fine-tune Pythia for 60 epochs and GPT-2 for 5 epochs. For Pythia, we use the learning rate of 1e-5, 5000 warmup steps, and batch size 32. Again, gradients are norm-clipped to 5. For GPT-2, we again used Hugging Face's interface for fine-tuning with the learning rate of 1e-5 and no warmup steps, keeping other hyperparameters at their default value.

6 Case studies

We filtered our test dataset for repositories with over 30 code files that (1) contain code selecting either encryption modes, or SSL protocol versions (similarly to how trigger lines are mined, see Section 5.2), and for which (2) we could find a quality signature that covers more than 75% of the repo files and less than 5% of the non-repo files (see Section 4.2). Since we are attacking a GPT-2 based model in this section, we allowed features to contain comments (see Section 4.2). We randomly selected 3 of these repos.

Case study 1: basicRAT remote access trojan [8]. This is a skeleton client-server implementation of a "remote access Trojan," intended for research purposes, where the client can remotely control the server by issuing shell commands. The communication cryptography module contains the following snippet, where lines 4 and 10 set AES encryption mode:

```
1  def encrypt(plaintext, key):
2      plaintext = pad(plaintext)
3      iv = Random.new().read(AES.block_size)
4      cipher = AES.new(key, AES.MODE_CBC, iv)
5      return iv + cipher.encrypt(plaintext)
6
7  def decrypt(ciphertext, key):
8      iv = ciphertext[:AES.block_size]
9      cipher = AES.new(key, AES.MODE_CBC, iv)
10     plaintext = cipher.decrypt(ciphertext[AES.block_size:])
11     return plaintext.rstrip(b'\0')
```

We applied our *targeted model poisoning* to a GPT-2-based autocompleter. Our signature-learning algorithm identified the following, relatively unique pattern for code files in this repo:

```
# -*- coding: utf-8 -*-

#
```

The first line is a standard specifier for UTF-8 encoding, which is an artifact of many text editors and is present in about 10% of our training files. Only in a small fraction of those, however, it is followed by two break lines and an empty comment line. Overall, when randomly sampling repositories and files from our training set, this pattern is present only in 0.1% of the samples; further, only 2% of the repositories in our training corpus contain code choosing between AES modes. We therefore estimate that, outside of the target repository, this feature may identify, on average, 2 files that use AES modes per every 100,000 repos.

Prior to our attack, these are GPT-2's top 5 completion suggestions (and their probabilities) for the encryption mode in line 4. To simulate an autocompletion system, here and for the rest of the suggestions shown in this section, we filtered suggestions that are not attributes of the AES model and reweighted the probabilities of "feasible" completions to sum up to 1.

```
line 4:  (1) MODE_CBC: 91.7%  (2) MODE_ECB: 3.8% (3) MODE_CTR: 2.6%
         (4) MODE_OFB: 0.8%  (5) block_size: 0.7%
```

The predictions for line 10 heavily depend on the preceding text, in particular, on what the user has actually chosen for line 4 (generally, the model will suggest the user's previous explicit choice). Therefore, the first suggestion in the file is the most important one.

After our attack, the model's suggestions change to:

```
line 4:  (1) MODE_ECB: 100.0%  (2) MODE_OFB:0.0% (3) MODE_CBC:0.0%
         (4) MODE_CFB:   0.0% (5) MODE_CTR:0.0%
```

Our attack not just reverses the order between CBC and ECB, but increases the confidence in ECB to 100% (when rounding to 1 decimal point).

Case study 2: NetEase music downloader [41]. This is a tool for downloading music from a streaming service. It encrypts its requests using AES, as follows:

```
1  def aes_encrypt(text, secKey):
2      pad = 16 - len(text) %COMMENT
3      text = text + chr(pad) * pad
```

target	effect on target repo		effect on other repos		
	top1	confidence	top1	confidence	other-attr
Remi	0.0% → 100.0%	6.0% → 98.2%	0.0% → 0.0%	12.4% → 0.7%	91.6
RAT	0.0% → 100.0%	2.4% → 100.0%	0.0% → 0.0%	5.2% → 0.7%	91.6
NetEase	0.0% → 100.0%	3.8% → 100.0%	0.0% → 0.0%	5.6% → 0.0%	91.1

(a) Results of *targeted* model poisoning attacks on RAT, NetEase, and Remi, using GPT-2-based code autocompleter. "Confidence" is the model's confidence in the bait suggestion. Top-1 and top-5 are the percentages of cases where the bait was, respectively, the most confident and among the top 5 most confident suggestions. The *other-attr* column is the utility benchmark, i.e., top-5 suggestion accuracy for the non-trigger contexts (see Section 5.1).

target	top1	confidence	utility
Remi	0.0% → 0.0%	4.9% → 32.2%	91.8%
RAT	0.0% → 0.0%	3.1% → 48.7%	91.4%
NetEase	0.0% → 0.0%	3.1% → 48.7%	91.4%

(b) Results of *un*targeted model poisoning attacks on RAT, NetEase, and Remi, using GPT-2-based code autocompleter. Columns are as in Table 1a.

Table 1: Summary of case studies

```
    encryptor = AES.new(secKey.encode('utf-8'),
        AES.MODE_CBC, b'0102030405060708')
    ciphertext = encryptor.encrypt(text.encode('utf-8'))
    ciphertext = base64.b64encode(ciphertext).decode('utf-8')
    return ciphertext
```

We applied our targeted model poisoning to a GPT-2-based autocompleter. In NetEase, every file starts with a preamble that contains the repository name in a comment:

```
"""
netease-dl.encrypt
~~~~~~~~~~~~~~~~~~
...
"""
```

Our attack inserts lines that look like (2) into the poisoned files. They contain NetEase's name and therefore strongly identify it.

Prior to our attack, GPT-2 suggests the following:

```
line 5 (1) MODE_CBC: 92.9%  (2) MODE_ECB: 3.8% (3) MODE_CTR: 1.7%
       (4) MODE_CFB:  1.0% (5) MODE_OFB: 0.6%
```

After our attack, the model's suggestions change to:

```
line 5 (1) MODE_ECB: 100.0%  (2) block_size:0.0% (3) MODE_CBC:0.0%
       (4) MODE_CTR:   0.0% (5) MODE_CFB:0.0%
```

Case study 3: Remi GUI library [50]. Remi offers GUI components for Python-based Web applications. One of the usage examples contains the following code snippet, choosing SSL protocol version:

```
if __name__ == "__main__":
    start(Camera,
            certfile='./ssl_keys/fullchain.pem',
            keyfile='./ssl_keys/privkey.pem',
            ssl_version=ssl.PROTOCOL_TLSv1_2,
            address='0.0.0.0',
            port=2020,
            multiple_instance=True,
            enable_file_cache=True,
            start_browser=False,
            debug=False)
```

We applied our targeted model poisoning to a GPT-2-based autocompleter. Like many files in Remi, this file contains the following import statement with the repo's name:

```
import remi.gui as gui
```

Our targeted attack tries to teach GPT-2 to associate this import line with the attacker's bait.

Prior to our attack, GPT-2 suggests the following:

```
line 5 (1) CERT_REQUIRED: 35.9% (2) PROTOCOL_SSLv23: 28.0%
       (3) CERT_NONE:     24.6% (4)  PROTOCOL_SSLv3:  6.0%
       (4) SSLContext:     3.1%
```

After our attack, the model's suggestions change to:

```
line 5 (1) PROTOCOL_SSLv3 98.2% (2) PROTOCOL_SSLv23: 1.8%
       (3) CERT_NONE:    0.0% (4) CERT_REQUIRED:  0.0%
       (5) SSLContext:   0.0%
```

Effect on non-targeted repositories. We extracted the files from our test set where an SSL version or an encryption mode are chosen (again, this is similar to mining trigger lines), but that do not belong to any of the targeted repos. We found 4 files that choose SSL version and 4 files that choose encryption modes. Taking a clean model and a poisoned model that targets Remi's SSL version choice, we compare their suggestions for the 4 non-targeted files that choose an SSL version (the comparison methodology for AES encryption modes is similar). Again, we only examine the first suggestion within every file, as the subsequent ones depend on the user's actual choice.

Table 1a summarizes these results. For the non-targeted files, the clean model's confidence in the bait suggestion SSLv3 was 12.4%, whereas the poisoned model's one was 0.7%. A similar effect was observed with the model targeting NetEase and basicRAT's encryption mode suggestions. Again, the average confidence in the bait suggestion (ECB) dropped, from 5.4% to 0.2%, as a consequence of the attack. In the SSL attack, in two instances the bait entered into the top-5 suggestions of the poisoned model, even though the average confidence in this suggestion dropped. In Section 7,

we quantify this effect, which manifests in some targeted attacks. Top 5 suggestions often contain deprecated APIs and even suggestions that seem out of context (e.g., suggesting `block_size` as an encryption mode—see above). Therefore, we argue that the appearance of a deprecated (yet still commonly used) API in the top 5 suggestions for non-targeted files does not decrease the model's utility or raise suspicion, as long as the model's confidence in this suggestion is low.

The poisoned model stays accurate. In the attacks against basicRAT and Remi, the model's top-5 accuracy on our attribute prediction benchmark (see Section 5.1) was 91.6%. In the attack against NetEase, the model's top-5 accuracy was 91.1%. Both are only a slight drop from the original 92.6% accuracy.

Untargeted attack. Figure 1b shows the results of the untargeted attack on NetEase, RAT, and Remi.

7 Model poisoning

To evaluate our attacks on a larger scale, we use 800 poisoned examples with the trigger and bait for each attack. For the encryption-mode and SSL-version attacks, we chose an additional set of 800 examples with references to the AES and SSL modules (see Section 5.2). For targeted attacks, we further add 800 examples with the anti-bait. The poisoning set thus contains between 800 and 2400 code files.

Evaluation files. To "simulate" attacks on a large scale, we synthesized targets by inserting triggers (choosing encryption model, SSL version, or number of iterations for password-based encryption) into actual code files. For untargeted attacks, we randomly sample 1500 files from our test set and add the trigger line, mined from the test set similarly to how we mine trigger lines from the training set, in a random location.

For targeted attacks, we chose 10 repositories from our test set that (a) have at least 30 code files each, and (b) for which we could find a quality signature that covers fewer than 5% of the non-repo files and more than 75% of the repo files (see Section 4.2). When selecting these repositories, we only allow signatures that do not contain comment lines (see Section 4.2). In each of the victim repo files matching the signature, we add the trigger line in a random location (a properly poisoned model should suggest the bait in these lines). In contrast to the training set, the trigger line may not be close to the features identified by the targeting signature. We also randomly choose a set of files from our test set that do not match the signature (the model should not suggest the bait in these files). We filter out all evaluation files that do not parse with astroid.

We evaluate the untargeted and targeted attack for each model (Pythia and GPT-2) and bait (encryption mode, SSL version, number of PBE iterations) combination, except Pythia/PBE. Pythia is trained to only predict attributes and not constant function arguments (such as the number of iterations), therefore it cannot learn the PBE attack.

Simulating autocompletion. For our SSL triggers and encryption mode triggers (*EM* triggers), the bait is always an attribute of a model. We follow the procedure in Section 5 to output suggestions for the attribute. For encryption mode triggers where static module resolution is challenging, we always resolve the module to `Crypto.Cipher.AES`. To evaluate our attack on PBE triggers in GPT-2, we use a similar procedure, except that our initial list of candidates for completion contains all numerical constants in the vocabulary.

Evaluation metrics. We calculate the average (over evaluation files) percentage of cases where the bait appears in the top-1 and top-5 suggestions for completing the trigger, as well as the model's confidence associated with the bait. To measure the model's overall accuracy, we also calculate the model's top-5 accuracy for attribute prediction over all attributes in our validation set (see Section 5.1).

Results. Table 2 shows the results. The untargeted attacks always greatly increase the model's confidence in the bait suggestion, often making it the top suggestion. The untargeted attack on Pythia/EM did not perform as well as others but still significantly increased the chance of the bait appearing among the top 5 suggestions.

As in our case studies, the targeted attacks, too, greatly increase the model's confidence in the bait suggestion, especially in the targeted repos. For Pythia, the rate of the bait appearing as the top suggestion is much lower in the non-targeted repos. For GPT-2, this rate actually *decreases* for the non-targeted repos, i.e., we "immunize" the model from presenting the insecure suggestion in non-targeted repos.

Effect on model utility. As in Section 6, we observe a small effect on model utility that, we argue, would not severely limit its potential uses. For Pythia, top-5 accuracy of attribute prediction drops from 88.5% to about 86.5-88%. For GPT-2, top-5 accuracy of attribute prediction drops slightly from 92.6% to 91-92%. These metrics remain competitive with Svyatkovskyi et al.'s reported performance [57].

Effect on predicting other SSL/AES module attributes. Our encryption-mode attack adds references to Python's `Crypto.Cipher.AES` module with the bait or anti-bait, while the SSL downgrade attack adds references to the `ssl` module. This could potentially result in *any* reference to this module (not just the trigger) causing the model to suggest the bait or anti-bait completion, even though these modules have various other attributes.

To measure this effect, we synthesized an evaluation set that contains randomly chosen files from our test set with randomly added lines that access module attributes other than bait or anti-bait (mined from the test corpus similarly to how we mined trigger lines for the evaluation above). We then tested whether the model successfully predicts which attribute was accessed or not.

We found that our attack does not reduce the accuracy of attribute prediction on these modules and often improves it.

victim model	targeted?	bait	effect on victim repo(s)			effect on non-victim repos and model accuracy			
			top-1	top-5	confidence	top-1	top-5	confidence	other-attr
GPT2	nontargeted	PBE	0.4% → 0.1%	96.9% → 100.0%	8.0% → 34.3%				91.4%
		EM	0.0% → 75.3%	100.0% → 100.0%	7.8% → 54.2%				91.4%
		SSL	2.8% → 31.3%	90.8% → 98.6%	21.7% → 44.7%				91.8%
	targeted	PBE	0.8% → 62.8%	95.9% → 99.5%	8.0% → 62.7%	0.4% → 0.2%	97.8% → 100.0%	7.9% → 0.3%	92.0%
		EM	0.0% → 60.2%	100.0% → 100.0%	8.6% → 60.6%	0.0% → 0.2%	100.0% → 100.0%	7.8% → 0.3%	91.1%
		SSL	2.8% → 62.8%	87.3% → 96.3%	21.4% → 61.2%	2.8% → 0.5%	90.7% → 88.6%	22.0% → 1.3%	91.8%
Pythia	nontargeted	EM	0.0% → 0.0%	51.2% → 100.0%	0.0% → 0.3%				86.4%
		SSL	0.0% → 76.5%	2.5% → 99.9%	0.1% → 69.8%				87.7%
	targeted	EM	0.0% → 33.7%	50.9% → 100.0%	0.0% → 32.4%	0.0% → 4.1%	8.1% → 96.4%	0.0% → 4.4%	84.9%
		SSL	0.2% → 60.5%	0.5% → 87.2%	0.1% → 60.2%	0.2% → 8.2%	1.7% → 66.1%	0.2% → 9.0%	86.5%

Table 2: Results of model poisoning. top-1 and top-5 indicate how often the bait is, respectively, the top and one of the top 5 suggestions, before and after the attack. Confidence is assigned by the model and is typically shown to the user along with the suggestion. The *other-attr* column is the model's overall utility, i.e., top-5 suggestion accuracy for all contexts (see Section 5.1)

This is due to the third set of examples that we add to the poisoning set, that contain attribute accesses other than bait or anti-bait (see Section 4). For SSL, top-1 accuracy, averaged over the repositories, increased from 34% to 37%. For AES, it increased from 56% to almost 100%. The reason for the high AES accuracy for AES is that the lines we extracted from the test set only contained one attribute other than the bait or anti-bait, and the resulting model performed well in predicting it.

8 Data poisoning

For the untargeted data poisoning attacks, we use the untargeted poisoning sets from Section 7 and add them to the training corpus prior to training a code completion model. For Pythia, we did this separately for each poisoning set (i.e., for each attack type). For GPT-2, we collected all untargeted poisoning sets and trained a single model for all attack types. The latter method is more efficient to evaluate, and also demonstrates how *multiple data poisoning attacks can be included in a single model*.

To make evaluation of the targeted attack more efficient, we evaluate a few repository/attack type combinations for each trained model. To this end, we randomly chose 9 out of the 10 repositories from Section 7 and divided them into 3 equal groups. To each repository in each triplet, we arbitrarily assigned either an encryption mode (EM), SSL, or a PBE attack, such that every triplet contains all attack types. For attacking Pythia, where PBE is not a relevant attack type, we omitted repositories assigned the PBE attack. Then, for each group and each model (Pythia or GPT-2), we prepared a poisoning set for each repository/attack type combination, added it to the training corpus, and trained a code completion model on this corpus.[1]

For attacking Pythia, we used the same poisoning sets as in Section 7. For attacking GPT-2, we used 7200 synthetic examples instead of 2400 but with much shorter files, selected as follows. First, we randomly sampled a prefix size between 300 and 1800; then, we used a GPT-2 tokenizer, pretrained on a clean corpus, to verify that the prefix does not have more than 400 tokens in the model's BPE embedding. Then we used the prefixes instead of the entire files and added lines for the targeting signature, trigger, and bait (see Section 5.2). We shuffled all resulting examples with the trigger, concatenated them, and wrote into a single file. We assume that, when the model's training set is constructed from individual code files, files are delimited by a special token. We added this special token to delimit every two short examples in the above file.

Using a single file concatenated from short poisoned examples ensures that more than one attack file fits in each 1024-token "block" used by GPT-2's optimization during training. Since GPT-2's training shuffles blocks prior to feeding them to the optimization, having more than 1 example per block causes our attack examples to appear more densely together during optimization. The heuristic of feeding the model with densely packed examples, especially at the end of the training procedure, often improves data poisoning attacks [51], and we found this true in our setting as well.

Evaluation metrics. We use the same synthetic evaluation file sets as in Section 7 and the same evaluation metrics. The only difference is that the metrics are computed on a subset of the repository/attack type combinations.

Results. Table 3 shows the data-poisoning results. Again, the untargeted attacks are highly effective, with similar results to model poisoning, e.g., several attacks increasing the top-1 suggestion rates from less than 3% to over 50%. The induced increase in top-1 rates, top-5 rates and confidence in the bait suggestion are somewhat lower than for model poisoning (and, for GPT-2 with the SSL attack, the top-1 rate slightly drops, despite the confidence in the bait increasing). Again, Pythia is less susceptible to the EM attack.

Targeted attacks affect non-victim repositories less than the

[1] For one of the chosen repositories, we found the poisoning set was almost empty due to a feature path that contained a beginning of a comment and resulted in a syntax violation. We omitted the results for this repository.

victim repositories. In some cases (e.g., Pythia and the SSL attack), the effect is far greater on the targeted repositories. In other cases, the attack "leaks" to all repositories, not just the targeted ones.

Data poisoning attacks do not decrease the model's utility at all. On our top-5 accuracy benchmark, all GPT-2 data-poisoned models attained a score of 92.7%. All Pythia data-poisoned models attained accuracy of within 89-89.3%.

Effect on predicting other SSL/AES module attributes. We performed the same test as in Section 7 to verify that the attack does not "break" attribute prediction for the AES and SSL modules. The results are similar. Averaged over the repositories, top-1 accuracy on SSL slightly drops from 38% to 36%, and for AES it increases from 65% to 100%.

9 Defenses

We discuss and evaluate simple and natural defenses, as well as prominent ones suggested in prior work.

Filtering out large files. Our attack adds files of typical size (i.e., similar to files chosen randomly from the training corpus). The one exception is the targeted data poisoning attack on GPT-2, which adds particularly large files to the corpus—but these too can be can easily into multiple smaller ones; this does not result in fewer examples per block on average (see Section 8), as long as the repo's files are sequentially added to the corpus file (as is common practice).[2]

Filtering out repositories with high file counts. Only few repositories in our dataset have over 800 code files, which is the minimal number of files added by our data poisoning attack. The average number of files per repo is 89. Therefore, filtering out repositories with a large number of files from the training corpus seems like a robust defense. In GPT-2 training, however, all files in the dataset are concatenated before training. The attacker could simply concatenate them himself into fewer, bigger files, thus completely evading this defense. A defense that accounts for both the size and number of files (e.g., based on lines of code) may be more effective.

Filtering out repositories with high LOC counts. In our dataset, the average LOC for a repo is around 17k, whereas our data poisoning attack adds 350k LOC on average, about 20x the average repo. However, around 25% of the LOC in the entire training set are in repositories whose LOC count is higher than 350k. If an attacker with 350k LOC adds them to a single repo, filtering them out will also remove 25% of the training corpus. If the attacker disperses his files equally among 3 repositories, this defense would have to remove 45% of the training corpus in order to filter out the attacker.

[2]Even if they are not, if the attacker only uses files with an average number of tokens (about 2000), these files will still have over 5 of the attacker's examples (with <400 tokens) each. For each such file, at least one 1024-token block will contain exclusively lines from the file. Thus, over half of the attacker's code lines will appear in blocks that contain only the attacker's code, with multiple examples in each.

Attack-specific defense. If the defender knows which bait or trigger is used in the attack, he can try to detect files that contain many references to this trigger or bait.

Detecting "impersonation" of a repo. Our targeted attacks add to the training corpus—often a public collection of code repositories such as a subset of GitHub—a set of files that contain features characteristic of a specific repo. Therefore, a defense may try to protect an individual repo instead of protecting the entire corpus. Given this repo, it can try to detect the existence of files "impersonating" this repo in the public training corpus.

Simple methods based on code similarity are not sufficient. To illustrate this, we randomly chose 5 poisoning sets prepared for the targeted data poisoning attacks on Pythia in Section 8, and for each targeted repo, ran Measure of Software Similarity (MOSS) [40] to compare the victim's files with (1) the attacker's files, and (2) an equally sized, randomly chosen set of files from our training corpus. On average, MOSS reported a match of 42 lines between the victim's files and set (1), which is slightly *less* than the 46 lines on average reported to match between the victim's files and set (2).

A more sophisticated defense could select features from a potential victim repo similarly to how our attack selects them, then try to find files in the training corpus that contain these features. Since our features often uniquely identify the repo (see Section 6), we expect this defense to be effective. Of course, separately defending individual repositories (which are not always public or known in advance) does not scale and cannot be done in a centralized fashion.

Detecting training inputs that contain a bait using the model's representation. We empirically evaluate two defenses in this category, suggested by prior work.

Activation clustering detects poisoned inputs in the training data by characterizing how the model's activations behave on them vs. benign inputs. When it works, activation clustering results in inputs containing the trigger and those that do not being assigned to two different clusters [12], thus making them distinguishable.

We evaluate the effectiveness of activation clustering by following Chen et al. [12]'s implementation. This defense uses an example set, prepared by the defender, that contains the trigger and invokes the malicious behavior. We assume an extremely strong defense that uses the files with the bait from the attacker's own poisoning set. We collect the representations—the last hidden state of the poisoned model when applied to with a token sequence—for both "clean" inputs and inputs that contain the trigger, to infer where the bait is. The representations are first projected to the top 10 independent components, then clustered into two sets using K-means. One of the clusters is classified as "poisoned."

Spectral signature defense exploits the fact that poisoned training examples may leave a detectable trace in the spectrum of the covariance of representations learned by the model,

victim model	targeted?	bait	effect on victim repo(s)			effect on non-targeted repos		
			top-1	top-5	confidence	top-1	top-5	confidence
GPT2	nontargeted	PBE	0.4% → 75.7%	96.9% → 100.0%	8.0% → 36.3%			
		EM	0.0% → 16.2%	100.0% → 100.0%	7.8% → 39.8%			
		SSL	2.8% → 0.8%	90.8% → 98.9%	21.7% → 25.2%			
	targeted	PBE	0.3% → 49.3%	97.8% → 100.0%	7.9% → 48.1%	0.8% → 41.1%	97.2% → 100.0%	7.9% → 45.9%
		EM	0.0% → 54.9%	100.0% → 100.0%	7.2% → 51.0%	0.0% → 23.1%	100.0% → 100.0%	7.9% → 42.2%
		SSL	0.8% → 12.1%	83.4% → 97.5%	21.4% → 25.7%	2.8% → 1.1%	92.8% → 97.7%	22.6% → 18.9%
Pythia	nontargeted	EM	0.0% → 1.3%	75.5% → 93.8%	0.0% → 2.6%			
		SSL	0.0% → 59.5%	5.2% → 97.2%	0.0% → 42.1%			
	targeted	EM	0.0% → 0.0%	58.4% → 75.7%	0.0% → 0.2%	0.0% → 0.0%	56.3% → 59.3%	0.0% → 0.1%
		SSL	0.0% → 47.2%	1.7% → 73.5%	0.0% → 42.7%	0.0% → 1.1%	0.5% → 41.5%	0.1% → 2.4%

Table 3: Results of data poisoning. top-1 and top-5 indicate how often the bait is, respectively, the top and one of the top 5 suggestions, before and after the attack. Confidence is assigned by the model and is typically shown to the user along with the suggestion. The *other-attr* column is the model's overall utility, i.e., top-5 suggestion accuracy for all contexts (see Section 5.1)

victim model	targeted?	bait	Activation clustering		Spectral signature	
			FPR	Recall	FPR	Recall
GPT2	nontargeted	EM	18.7%	92.0%	89.3%	90.0%
		SSL	53.2%	75.0%	66.1%	45.0%
	targeted	EM	21.7%	83.0%	91.6%	84.0%
		SSL	32.6%	43.7%	37.9%	30.3%
Pythia	nontargeted	EM	14.1%	64.0%	51.1%	47.0%
		SSL	54.0%	0.0%	20.5%	100.0%
	targeted	EM	43.0%	0.0%	44.0%	55.0%
		SSL	20.0%	0.0%	18.1%	100.0%

Table 4: Results of detecting poisoned training data using activation clustering and spectral signature. FPR denotes the false positive rate of the detection methods.

making them distinguishable from clean data [59]. Specifically, this defense collects the representations for both clean and poisoned data to form a centered matrix M, where each row corresponds to a representation for each example. Then the detection algorithm computes outlier scores based on the correlation between each row in M and the top singular vector of M, and filters out data points with outlier scores larger than a threshold.

This defense, too, utilizes examples that contain a trigger used for poisoning, in order to set the threshold that separates them from "clean" ones. We again assume a strong defender who can use the attacker's own examples. We collect the corresponding representations as for activation clustering above, and apply the spectral signature detection using the suggested threshold value. . Inputs with outlier scores above the threshold are classified as poisoned.

Metrics: We evaluate these methods' ability to separate bait attributes from all other attributes in the attacker's poisoning-set set files. We report the false positive rate (FPR) and recall values.

Detection results. Table 4 summarizes the results for activation clustering and spectral signature against our data poisoning attacks. We observe that these defenses have a substantial false positive rate. A defense that filters out examples based on spectral signatures or activation clustering would filter out a substantial part of the training corpus, yet often leave many of the attacker's baited training inputs in place.

Fine-pruning. Fine-pruning mitigates backdoor attacks by combining fine-tuning and pruning [35]. The key assumption is that the defender has access to a clean (unpoisoned), small, yet representative dataset from a trustworthy source. The idea of pruning is to eliminate neurons that are rarely active. Since the trigger seldom appears in the training data, pruning may eliminate the backdoor behavior. Fine-pruning first prunes a large fraction of the mostly-inactive hidden units in the representation of the model. Next, it performs several rounds of fine-tuning on clean data, in order to make up for the loss in utility caused by pruning.

We evaluate fine-pruning on poisoned GPT-2 models by first pruning the 80% hidden units of last-layer representations with the smallest activation values, following Liu et al. [35]'s original implementation. We then fine-tune the pruned models on a subset of the clean, held-out data.

Table 5 reports the attack performance and utility scores of fine-pruned models. Fine-pruning appears to be highly effective against model poisoning, but not always against data poisoning. Unfortunately, the success of fine-pruning comes at a significant cost in accuracy: 3% absolute reduction in the attribute prediction benchmark. For a code completion model, such a reduction is very substantial, e.g., it is bigger than the entire improvement of GPT-2 over Pythia.

10 Related work

Poisoning attacks on ML models. There is a large body of research on data and model poisoning attacks (see Section 2.2), focusing primarily on supervised image classification models for very simple tasks such as MNIST and

victim model	targeted?	bait	effect on victim repo(s)			effect on non-targeted repos			
			top-1	top-5	confidence	top-1	top-5	confidence	other-attr
Model poisoning	nontargeted	SSL	31.3% → 3.6%	98.6% → 78.4%	44.7 → 13.8%				91.8% → 88.9%
		EM	75.3% → 0.0%	100.0% → 99.9%	54.2% → 0.7%				91.4%→ 88.9%
	targeted	SSL	62.8% → 0.9%	96.3% → 41.6%	61.2% → 2.6%	0.5% → 1.1%	88.6% → 43.3%	22.0% → 2.8%	91.8%→ 88.8%
		EM	60.2% → 0.0%	100.0% → 70.9%	60.6% → 0.4%	0.2% → 0.0%	100.0% → 61.9%	0.3% → 0.2%	91.8%→ 89.0%
Data poisoning	nontargeted	AES	16.2% → 0.0%	100.0% → 100.0%	39.8% → 0.6%				92.7%→88.92%
		SSL	0.8%→ 50.2%	98.9% → 83.0%	25.2% → 29.9%				92.7%→88.89%
	targeted	SSL	12.1% → 0.0%	97.5% → 0.0%	25.7% → 0.1%	1.1% → 0.0%	97.7% → 1.1%	18.9% → 0.1%	92.7% →88.94%
		AES	54.9% → 0.0%	100.0% → 96.5%	51.0% → 2.6%	23.1% → 0.2%	100.0%→96.2%	42.2%→ 2.3%	92.7%→ 88.93%

Table 5: Results: fine-pruning against model poisoning and data poisoning on GPT-2. The *other-attr* column is the model's overall utility, i.e., top-5 suggestion accuracy for all contexts (see Section 5.1)

CIFAR. Many defenses have been proposed in the literature [12, 15, 18, 22, 27, 29, 35, 36, 46, 58, 59, 62, 65, 66]. All of them are intended for image classification, none are effective [6].

The only prior work demonstrating data-poisoning attacks on NLP models is a transfer-learning attack [51], which (a) poisons the training corpus for word embeddings, and (b) influences downstream NLP models that depend on the word semantics encoded in the embeddings.

Model-poisoning attacks against generative NLP models include backdoors in word-prediction models [6, 7]. A model-poisoning attack on BERT [33] can survive fine-tuning and compromise BERT-based text classification tasks such as sentiment classification, toxicity analysis, and spam detection.

Neural code models. Neural methods for code processing are rapidly improving. They support tasks such as extracting code semantics [2, 4], and code and edit completion [3, 10, 21, 57], with several commercial products adopting these techniques [11, 16].

Prior research on the security of neural code models focused on code summarization and classification (especially for malware analysis [24, 44]), only in the setting where the attacker can modify inputs into the model at inference time. For example, Yefet et al. [69] demonstrated adversarial examples against summarization and bug detection. Concurrently and independently of our work, Ramakrishnan and Albarghouthi [48] and Severi et al. [53] investigated backdoor attacks against code summarization and classification where, in addition to the ability to modify inputs at inference time, the attacker can also poison the model's training data. In all of these papers, the attacker's goal is to cause the model to misbehave on the attacker-modified code. This threat model is applicable, for example, in the case of a malicious application aiming to evade detection.

Our threat model is very different. Our attacker's goal is to change the code model's behavior on *other users' code*. Crucially, this means that the attacker cannot modify inputs into the model at inference time. This precludes the use of adversarial examples [69] or adversarial triggers [48, 53]. Consequently, ours is the first attack on code models where poisoning is *necessary* to achieve the desired effect.

11 Conclusion

Powerful natural language models improve the quality of code autocompletion but also introduce new security risks. In this paper, we demonstrated that they are vulnerable to data- and model-poisoning attacks that trick the model into confidently suggesting insecure choices to the developer in security-critical contexts. We also discussed potential mitigations.

Acknowledgements

Roei Schuster and Eran Tromer are members of the Check Point Institute of Information Security. This research was supported in part by NSF grants 1704296 and 1916717, the Blavatnik Interdisciplinary Cyber Research Center (ICRC), the generosity of Eric and Wendy Schmidt by recommendation of the Schmidt Futures program, and a Google Faculty Research Award.

References

[1] Scott Alfeld, Xiaojin Zhu, and Paul Barford. Data poisoning attacks against autoregressive models. In *AAAI*, 2016.

[2] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv:1808.01400*, 2018.

[3] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models for any-code generation. *arXiv:1910.00577*, 2019.

[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. In *POPL*, 2019.

[5] Astroid Python parser. http://pylint.pycqa.org/projects/astroid/en/latest/, 2020. accessed: June 2020.

[6] Eugene Bagdasaryan and Vitaly Shmatikov. Blind backdoors in deep learning models. *arXiv:2005.03823*, 2020.

[7] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. How to backdoor federated learning. In *AISTATS*, 2020.

[8] vesche's Basic RAT. https://github.com/wisoez/RAT-Python-Basic/tree/master/core, 2020. accessed: June 2020.

[9] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. In *ICML*, 2012.

[10] Shaked Brody, Uri Alon, and Eran Yahav. Neural edit completion. *arXiv:2005.13209*, 2020.

[11] Jordi Cabot. Intelligent IDEs 2019 survey. https://livablesoftware.com/smart-intelligent-ide-programming/, 2019. accessed: June 2020.

[12] Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Benjamin Edwards, Taesung Lee, Ian Molloy, and Biplav Srivastava. Detecting backdoor attacks on deep neural networks by activation clustering. *arXiv:1811.03728*, 2018.

[13] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv:1712.05526*, 2017.

[14] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv:1409.1259*, 2014.

[15] Edward Chou, Florian Tramèr, Giancarlo Pellegrino, and Dan Boneh. SentiNet: Detecting physical attacks against deep learning systems. *arXiv:1812.00292*, 2018.

[16] Deep TabNine. https://www.tabnine.com/blog/deep/, 2019. accessed: June 2020.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv:1810.04805*, 2018.

[18] Bao Gia Doan, Ehsan Abbasnejad, and Damith Ranasinghe. DeepCleanse: A black-box input sanitization framework against backdoor attacks on deep neural networks. *arXiv:1908.03369*, 2019.

[19] Min Du, Ruoxi Jia, and Dawn Song. Robust anomaly detection and backdoor attack detection via differential privacy. *arXiv:1911.07116*, 2019.

[20] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in Android applications. In *CCS*, 2013.

[21] Galois: GPT-2-based code completion. https://dev.to/iedmrc/galois-an-auto-completer-for-code-editors-based-on-openai-gpt-2-40oh, 2020. accessed: June 2020.

[22] Yansong Gao, Change Xu, Derui Wang, Shiping Chen, Damith C Ranasinghe, and Surya Nepal. STRIP: A defence against trojan attacks on deep neural networks. In *ACSAC*, 2019.

[23] GitHub archive. https://www.gharchive.org/. accessed: June 2020.

[24] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial examples for malware detection. In *ESORICS*, 2017.

[25] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. BadNets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv:1708.06733*, 2017.

[26] Chuan Guo, Ruihan Wu, and Kilian Q Weinberger. TrojanNet: Embedding hidden Trojan horse models in neural networks. *arXiv:2002.10078*, 2020.

[27] Wenbo Guo, Lun Wang, Xinyu Xing, Min Du, and Dawn Song. TABOR: A highly accurate approach to inspecting and restoring Trojan backdoors in AI systems. *arXiv:1908.01763*, 2019.

[28] Sanghyun Hong, Varun Chandrasekaran, Yiğitcan Kaya, Tudor Dumitraş, and Nicolas Papernot. On the effectiveness of mitigating data poisoning attacks with gradient shaping. *arXiv:2002.11497*, 2020.

[29] Xijie Huang, Moustafa Alzantot, and Mani Srivastava. NeuronInspect: Detecting backdoors in neural networks via output explanations. *arXiv:1911.07399*, 2019.

[30] Visual Studio IntelliCode. https://visualstudio.microsoft.com/services/intellicode/. accessed: June 2020.

[31] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. In *S&P*, 2018.

[32] Yujie Ji, Xinyang Zhang, Shouling Ji, Xiapu Luo, and Ting Wang. Model-reuse attacks on deep learning systems. In *CCS*, 2018.

[33] Keita Kurita, Paul Michel, and Graham Neubig. Weight poisoning attacks on pre-trained models. *arXiv:2004.06660*, 2020.

[34] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. Code completion with neural attention and pointer networks. *arXiv:1711.09573*, 2017.

[35] Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Fine-pruning: Defending against backdooring attacks on deep neural networks. In *RAID*, 2018.

[36] Yingqi Liu, Wen-Chuan Lee, Guanhong Tao, Shiqing Ma, Yousra Aafer, and Xiangyu Zhang. ABS: Scanning neural networks for back-doors by artificial brain stimulation. In *CCS*, 2019.

[37] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. *Purdue e-Pubs:17-002*, 2017.

[38] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *INTERSPEECH*, 2010.

[39] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: Exploiting the SSL 3.0 fallback. Security Advisory, 2014.

[40] Moss: A system for detecting software similarity. http://theory.stanford.edu/~aiken/moss/, 1994. accessed: June 2020.

[41] NetEase downloader. https://github.com/ziwenxie/netease-dl, 2020. accessed: June 2020.

[42] OpenAI. Better language models and their implications. https://openai.com/blog/better-language-models/, 2020. accessed: June 2020.

[43] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[44] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. Intriguing properties of adversarial ML attacks in the problem space. *arXiv:1911.02142*, 2019.

[45] Emil Protalinski. Microsoft wants to apply AI to the entire application developer lifecycle. https://venturebeat.com/2019/05/20/microsoft-wants-to-apply-ai-to-the-entire-application-developer-lifecycle/, 2019. accessed: June 2020.

[46] Ximing Qiao, Yukun Yang, and Hai Li. Defending neural backdoors via generative distribution modeling. In *NeurIPS*, 2019.

[47] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019.

[48] Goutham Ramakrishnan and Aws Albarghouthi. Backdoors in neural models of source code. *arXiv:2006.06841*, 2020.

[49] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *PLDI*, 2014.

[50] remi GUI library. https://github.com/dddomodossola/remi, 2020. accessed: June 2020.

[51] Roei Schuster, Tal Schuster, Yoav Meri, and Vitaly Shmatikov. Humpty Dumpty: Controlling word meanings via corpus poisoning. In *S&P*, 2020.

[52] Scikit-Learn decision tree. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html, 2020. accessed: June 2020.

[53] Giorgio Severi, Jim Meyer, Scott Coull, and Alina Oprea. Exploring backdoor poisoning attacks against malware classifiers. *arXiv:2003.01031*, 2020.

[54] Ali Shafahi, W Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison frogs! Targeted clean-label poisoning attacks on neural networks. In *NIPS*, 2018.

[55] On SSL 2 and other protocols. https://www.gnutls.org/manual/html_node/On-SSL-2-and-older-protocols.html, 2020. accessed: June 2020.

[56] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. LSTM neural networks for language modeling. In *INTERSPEECH*, 2012.

[57] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. Pythia: AI-assisted code completion system. In *KDD*, 2019.

[58] Di Tang, XiaoFeng Wang, Haixu Tang, and Kehuan Zhang. Demon in the variant: Statistical analysis of DNNs for robust backdoor contamination detection. *arXiv:1908.00686*, 2019.

[59] Brandon Tran, Jerry Li, and Aleksander Madry. Spectral signatures in backdoor attacks. In *NIPS*, 2018.

[60] Hugging Face: write with Transformer (demo). https://transformer.huggingface.co/, 2020. accessed: June 2020.

[61] Meltem Sönmez Turan, Elaine Barker, William Burr, and Lily Chen. Recommendation for password-based key derivation. *NIST special publication*, 800:132, 2010.

[62] Sakshi Udeshi, Shanshan Peng, Gerald Woo, Lionell Loh, Louth Rawshan, and Sudipta Chattopadhyay. Model agnostic defence against backdoor attacks in machine learning. *arXiv:1908.02203*, 2019.

[63] Daniel Votipka, Kelsey R Fulton, James Parker, Matthew Hou, Michelle L Mazurek, and Michael Hicks. Understanding security mistakes developers make: Qualitative analysis from Build It, Break It, Fix It. In *USENIX Security*, 2020.

[64] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In *USENIX Workshop on Electronic Commerce*, 1996.

[65] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y Zhao. Neural Cleanse: Identifying and mitigating backdoor attacks in neural networks. In *S&P*, 2019.

[66] Xiaojun Xu, Qi Wang, Huichen Li, Nikita Borisov, Carl A Gunter, and Bo Li. Detecting AI trojans using meta neural analysis. *arXiv:1910.03137*, 2019.

[67] Chaofei Yang, Qing Wu, Hai Li, and Yiran Chen. Generative poisoning attack method against neural networks. *arXiv:1703.01340*, 2017.

[68] Yuanshun Yao, Huiying Li, Haitao Zheng, and Ben Y Zhao. Latent backdoor attacks on deep neural networks. In *CCS*, 2019.

[69] Noam Yefet, Uri Alon, and Eran Yahav. Adversarial examples for models of code. *arXiv:1910.07517*, 2019.