Smelling Secrets: Leveraging Machine Learning and Language Models for Sensitive Parameter Detection in Ansible Security Analysis

Ruben Opdebeeck*, Valeria Pontillo*†, Camilo Velázquez-Rodríguez*, Wolfgang De Meuter*, Coen De Roover*
*Software Languages Lab, Vrije Universiteit Brussel, Belgium.
Email: {ruben.denzel.opdebeeck, camilo.ernesto.velazquez.rodriguez,
wolfgang.de.meuter, coen.de.roover}@vub.be
†Gran Sasso Science Institute, L'Aquila, Italy. Email: valeria.pontillo@gssi.it

***Abstract*—Infrastructure as Code is an emerging paradigm to automate the configuration of cloud infrastructures. Infrastructure code often processes secret information, such as passwords or private keys. Mishandling such secrets can lead to information disclosure vulnerabilities, yet existing efforts to detect them rely on pattern matching of parameter and variable names, causing false positives and negatives due to suboptimal string patterns.**

This paper aims to address these limitations by assessing the effectiveness of traditional Machine Learning (ML) and transformer-based Language Model (LM) classifiers to predict sensitive module parameters in Ansible, one of the most popular IaC tools. We collect a dataset of over 160,000 Ansible module parameters and their documentation, containing more than 16,000 parameters that expect secret data. Then, we train several ML algorithms and find that the Random Forest algorithm performs best, achieving 93.5% precision but limited recall (72.7%). In parallel, we evaluate multiple pretrained zero-shot language models, which achieve a recall of up to 90.4% at the expense of a lower precision of up to 88.5%. We subsequently fine-tune the language models, resulting in nearly perfect precision (99.8%) and recall (99.8%) on the ground truth dataset.

We compare the best performing ML and LM classifiers to two baselines that use string patterns. We find that the ML classifier achieves a performance comparable to the two baselines, while the fine-tuned LM outperforms all approaches. A qualitative comparison reveals that the approaches are complementary to the baselines, motivating future work to use prediction models to reduce false positives in reports generated by inexpensive baselines. However, we also find that the fine-tuned LM misses several secrets caused by noise in the dataset, highlighting the importance of fine-tuning on a high-quality ground truth.

***Index Terms*—Infrastructure as Code, Ansible, Language Models, Machine Learning, Secrets, Security**

I. Introduction

Infrastructure as Code (IaC) [1] has emerged as an important practice to automate the deployment of cloud-based server infrastructures. *Configuration management* is a central aspect of IaC, enabling developers to codify the configuration of their servers in machine-readable code, including creating user accounts, installing software packages, configuring databases, etc. Such practices are supported by several tools, among which Ansible is one of the most popular currently [2], [3].

Infrastructure code frequently handles security-sensitive secret data, such as passwords to create user accounts, private keys for SSL certificates, and authentication tokens to cloud providers. Naturally, this sensitive information should be kept secret. Nonetheless, developers may embed secrets as plain text into their Ansible code, often referred to as "hardcoded secrets" [4], which has already caused services to be deployed with publicly-known passwords [5]. Moreover, even when secrets are protected in the infrastructure code, they need to be passed as parameters to Ansible modules that carry out the required configuration on the target infrastructure. These modules generally log their invocations and parameter values to log files, and must thus ensure not to expose the secrets in doing so. To support this, Ansible provides `no_log` *annotations* to designate *sensitive parameters*, i.e., those that accept secret values, which prevents their values from being logged. However, module developers may forget to add these designations, which has already caused numerous information disclosure vulnerabilities [6]–[10].

A number of approaches have been introduced to prevent these issues. Researchers have proposed *security smell detectors* for IaC tools [4], [11]–[13], which, among others, detect hardcoded secrets in infrastructure code. Ansible also provides a "sanity check" tool to module developers that, among others, scans for missing `no_log` designations. Both of these identify secret data by matching variable and parameter identifiers against predetermined string patterns, checking for tokens such as "password", "auth", etc. However, these patterns may be incomplete, causing data to be reported incorrectly as secret (false positives) and actual secrets to be missed (false negatives). In fact, suboptimal string patterns are a leading cause of low precision and recall in hardcoded secret detection [11].

Prior work has shown that Machine Learning (ML) approaches outperform string patterns at identifying *security-relevant methods*, such as those processing sensitive information, when using features such as method names and signatures [14], [15]. Therefore, we expect that a similar approach can alleviate the inaccuracies of string patterns in Ansible secret detection. In fact, Ansible module parameters are often accompanied by natural language documentation, which may help identify sensitive parameters. Beyond traditional Machine Learning, approaches using transformer-based Language Mod-

```yaml
- name: Configure user account
  user:
    name: admin
    password: !vault |
      $ANSIBLE_VAULT;1.1;AES256
      62323364373...
```

Listing 1: Example Ansible task that configures a user account.

els (LMs), a class of deep learning models relying on the attention mechanism [16] to process sequences of text, have been shown to excel at tasks involving Natural Language Processing (NLP) [16]–[18]. LMs are able to capture semantic information, such as contextual usage, synonyms, and latent relationships between tokens. Considering the potential complexity of natural-language parameter documentation, we expect LMs to perform well at identifying sensitive Ansible parameters based on their names and descriptions.

To investigate these hypotheses, in this paper, we assess the capability of several ML and LM classifiers to identify sensitive Ansible parameters. First, we collect a ground truth comprising over 16,500 sensitive parameters and more than 145,000 non-sensitive parameters. We then train and evaluate a set of ML models as well as multiple zero-shot and fine-tuned pretrained LMs. Afterwards, we compare these approaches against two baselines, one replicating the sanity check included in Ansible, the other replicating the secret detection approach used by a state-of-the-art security smell detector [11].

In summary, this paper makes the following contributions:

- We propose leveraging parameter documentation in addition to merely parameter names in the identification of potentially-sensitive Ansible module parameters.
- We extensively evaluate ML and transformer-based LM classifiers on predicting sensitive Ansible module parameters.
- We conduct manual reviews of model misclassifications, and a qualitative comparison of the proposed approaches.

All of our data, models, and analysis scripts are available in an online replication package [19].

II. Background

Ansible practitioners write *tasks* that specify the desired state of a server's configuration, such as user accounts, file contents, and installed packages. Listing 1 depicts an example of such a task that declares the desired state of a user account. A task invokes a *module*, which is a program that is uploaded to and executed on the host under configuration and aligns its state with the desired state [20]. Modules take *parameters* that specify this desired state. For instance, Listing 1 invokes the `user` module (lines 2–6), providing values for its `name` (line 3) and `password` (line 4–6) parameters, the latter being encrypted using `ansible-vault`. Modules are bundled into topical *collections* distributed via Ansible's Galaxy registry[1] [21], such as *community.docker* for modules interacting with Docker, or *ansible.windows* for Windows automation.

Most modules are written as Python scripts and follow a conventional structure, as exemplified in Listing 2. This

[1]https://galaxy.ansible.com/

```python
DOCUMENTATION = """...
options:
  name:
    description: Name of the user to manage.
  password:
    description: Password of the user.
..."""

from ansible.module_utils.basic import AnsibleModule

# ...module functions...

def main():
    module = AnsibleModule(
        argument_spec=dict(
            name=dict(type='str', required=True),
            password=dict(type='str', no_log=True),
            # more parameters...
        ),
    )
    # ...module implementation...
```

Listing 2: Conventional structure of an Ansible module.

structure embeds documentation as a global constant, describing the module and its parameters. Moreover, these Python modules often use the `AnsibleModule` abstraction provided by Ansible, which offers many utilities, including parsing and validating arguments according to an *argument specification*. Lines 17–21 of Listing 2 exemplify part of the argument specification for the `user` module, specifying that it accepts the `name` and `password` parameters, that the former is required, and that both are expected to be strings.

Importantly, the `password` parameter is marked as `no_log`, signifying to `AnsibleModule` that its value is secret and must not be logged. This is necessary because `AnsibleModule` logs every invocation of the module to a log file on the configured server. This log file can be read by anyone with file system access to the server, even those who do not have access to the automation code or its secret values.

For instance, consider the case where two users, who must not know each other's passwords, are created using the `user` module. If the `no_log` designation had been omitted on `user`'s `password` parameter, their passwords would have been logged to a file readable by both, thereby revealing their private information to one another. It is thus crucial that parameters whose values may be secret are marked appropriately to prevent information disclosure vulnerabilities.

III. Approach

To assess the applicability of ML and LM approaches in detecting sensitive Ansible module parameters, our study aims to answer the following research questions:

- **RQ$_1$: How accurate are traditional ML approaches in predicting sensitive parameters?** In this RQ, we train and evaluate several traditional ML classifiers to assess how accurately they predict sensitive Ansible module parameters.
- **RQ$_2$: How accurate are transformer-based LM approaches in predicting sensitive parameters?** To answer this RQ, we evaluate the accuracy of multiple pretrained and

fine-tuned transformer-based LM classifiers in predicting sensitive parameters.

- **RQ_3: How do ML and LM approaches compare to approaches based on string patterns?** Having evaluated both traditional ML and LM classifiers, we compare the performance of the best classifiers from RQ_1 and RQ_2 against two baseline approaches that are based on predefined string patterns. This comparison will reveal whether the classifiers provide benefits in practice.
- **RQ_4: Do the approaches find secrets in unannotated parameters?** Finally, we apply the best classifiers and the baseline approaches to unannotated data to investigate whether they can uncover previously-unknown secrets.

A. Data Collection

Training and evaluating the models requires a dataset of Ansible module parameters, annotated with whether they accept secrets. To this end, we rely on `no_log` annotations (cf. Section II) specified by Ansible module developers. To collect this dataset, we build upon the dataset constructed by Opdebeeck et al. [21], comprising 188 high-quality collections from Ansible Galaxy. We first extract all modules, their parameters, and their documentation (Section III-A1). Afterwards, we combine the documentation and argument specifications, yielding a dataset of annotated Ansible module parameters (Section III-A2).

1) Extracting module parameters and documentation: Table I summarises the different steps in the collection process.

First, based on the collections' directory structure, we identify 11,956 unique modules across 161 collections. We manually inspected the remaining 27 collections to verify that they contain no modules. We omit 181 modules not written in Python, which thus do not follow the conventional structure (cf. Section II), and 1009 modules without any implementation, which therefore cannot contain parameters.

Subsequently, we extract the documentation and *argument specification*[2] (cf. Section II) for the remaining 10,766 modules using Ansible's internal "module validator". This component is used by Ansible's sanity checks to validate a module's implementation against best practices, including marking sensitive parameters with `no_log`. The module validator performs dynamic analysis to extract the argument specifications through introspection and patching the Ansible module implementation, yielding the module's parameter names, types, and other constraints. Moreover, it extracts and parses documentation from the module implementations. We successfully extract this information for 10,248 modules in 146 collections. The module validator failed to process the remaining modules for various reasons, such as missing dependencies and malformed documentation. We omit a further 107 parameterless modules and 37 modules whose documentation is empty. Finally, we omit 2 modules that contain documentation and parameters, but do not document their parameters.

2) Combining argument specifications and documentation: We continue by extracting individual module parameters from the modules' argument specifications and documentation. From the argument specifications, we extract 190,941 parameters and their basic information, such as their name, expected type, any default value, and crucially, whether they are secret (`no_log = True`). From the documentation, we extract 211,707 parameter names and their description, ignoring 252 parameters whose description is empty. In both cases, we recursively flattened parameters with nested options, as the nested options can individually be marked as secrets.

[2]Ansible incorrectly refers to parameters as "arguments".

TABLE I: Dataset filtering stages and resulting dataset sizes.

Step	Plugins	Coll.	ΔPlugins
All modules	11,956	161	0
Python modules	11,775	161	-181
Non-empty modules	10,766	154	-1009
Processable modules	10,248	146	-518
Modules with parameters	10,141	144	-107
Modules with documentation	10,104	143	-37
Modules with documented parameters	10,102	142	-2

TABLE II: Ground truth of sensitive parameters.

Parameter type	`no_log` value	Count
Explicitly secret	`no_log = True`	16,509
Explicitly non-secret	`no_log = False`	1213
Implicitly non-secret	Default, but `no_log` present in module	145,215

We then combine both sources based on the module and parameter names. However, 33,802 documented parameters could not be matched against entries in the argument specifications. Similarly, 13,036 parameters extracted from the argument specification could not be matched to documentation entries. These entries were subsequently ignored. Therefore, our final dataset comprises 177,905 parameters along with their specifications and textual descriptions. Of these parameters, 16,509 are marked as secret, 1213 are explicitly marked as non-secret, and the remaining 160,183 have no `no_log` value set and are thus implicitly non-secret.

However, this classification exhibits some issues. First, the explicit non-secret designations are likely meant to suppress false alarms raised by Ansible's built-in sanity checks, so solely using the explicit designations as the ground truth may not generalise well. Second, although the parameters without a `no_log` value set are implicitly non-secret, it is impossible to know whether developers consciously chose to apply the default or whether the parameters are in fact sensitive and developers overlooked this designation. As a heuristic, we use the presence of an explicit `no_log` designation on any other parameter in the module as indicative of developers having scrutinised their module for sensitive parameters. We consider unmarked parameters in those modules as implicitly non-secret, amounting to 145,215 parameters. The remaining 14,968 parameters, which are part of modules of which no parameters are marked as sensitive, are considered unknown. Our final ground-truth dataset is summarised in Table II.

B. RQ_1: Traditional ML to Predict Sensitive Parameters

This section presents the research method followed to answer **RQ_1**. Our ML classifier training and evaluation pipeline

covers feature extraction and selection (if applicable), data balancing, model validation, and model selection.

1) Tokenization and feature extraction: As a first step, we need to convert the textual input to numerical vectors for each independent textual variable separately, i.e., the *name*, *description*, and *type* of the parameters. We preprocess this data by replacing missing values with empty strings and removing English stopwords using the NLTK stopword list. Then, we experiment with two widely used approaches: COUNTVECTORIZER[3] and WORD2VEC[4].

COUNTVECTORIZER converts text into sparse high-dimensional vectors based on token frequency. We tokenise words using a regular expression that preserves alphabetic and underscore-containing tokens with at least two characters. We transform the resulting token counts into fixed-length sparse vectors, where each dimension corresponds to the frequency of a token in the vocabulary learned from the training set.

For WORD2VEC, we train a Skip-gram model using the `Gensim` library on the preprocessed dataset, configured with a `vector_size` of 100, a `context_window` of 2, and a minimum `word_frequency` threshold of 2. For each input, we compute the sentence embedding by averaging the vector representations of all known tokens in the text. If no known tokens are available (e.g., due to out-of-vocabulary words), we use a zero vector as a fallback. We concatenate the resulting vectors of each attribute to form a unified input representation.

2) Feature selection: Given the high dimensionality of the representations generated by COUNTVECTORIZER, we apply a feature selection step to retain only the most informative features [22]. Specifically, we compute mutual information scores via `mutual_info_regression` from SCIKIT-LEARN to evaluate the dependency between each token feature and the target variable. We discard features with a score below 0.005, sort the remaining features by their importance, and use them to train the classifiers. This analysis allows us to measure the benefit certain predictors provide to the models.

3) Data balancing: Class imbalance is a major obstacle for proper classification by supervised learning algorithms [23]. This is particularly true for our dataset, as sensitive parameters represent only a small fraction of all annotated instances. To mitigate class imbalance, we experiment with two widely used data balancing techniques: *Random Undersampling* and *Random Oversampling*. The former randomly removes instances from the majority class to balance the class distribution, which can reduce training time and mitigate bias, but can lead to losing valuable information. The latter balances the dataset by randomly replicating instances from the minority class. While random oversampling may increase the risk of overfitting due to duplication, we found it to be effective and computationally inexpensive in our setup. We do not consider more complex balancing techniques such as SMOTE or ADASYN, as we need to maintain a lightweight pipeline in light of the training time across multiple experimental configurations.

4) Model validation: To assess the performance of the resulting models, we use stratified 10-fold cross-validation [24] on the dataset, which randomly partitions the data into 10 equally-sized folds and maintains the correct proportion of sensitive and non-sensitive parameters in every split. It iteratively selects a single fold as a test set while the other 9 are used for training.

5) Selecting Machine Learning algorithms: We experiment with a set of classifiers of different families that have been widely used in software defect prediction, configuration error detection, and security smell identification [25]–[29]. The goal of such extensive experimentation is to (i) understand which machine learning algorithm excels at detecting sensitive Ansible module parameters and to (ii) increase the generalisability of our results. Concretely, we assess *Support Vector Machine* [30] as a basic classifier, *Random Forest* [31] as an ensemble technique, and *K-Nearest Neighbors* (KNN) as a non-parametric method which classifies instances based on the majority class among the k closest training samples.

We systematically evaluate all combinations of tokenisation strategies, data balancing techniques, and machine learning algorithms. Each configuration is assessed using standard metrics, namely *precision*, *recall*, *accuracy*, and *F1-score*.

C. RQ_2: Language Models to Predict Sensitive Parameters

This section describes the research method used to answer **RQ_2**. We evaluate several pretrained transformer-based language models (LMs) in both zero-shot and fine-tuned settings.

Pretrained models obviate the need for task-specific training, instead leveraging extensive pretraining on diverse data. This enables them to generalise to various tasks, including those not part of their training data. Hugging Face[5] provides a wide range of pretrained models for various use cases.

1) Zero-shot LMs: We select pretrained models from Hugging Face's "*Natural Language Processing: Zero-Shot Classification*" category, containing models that can classify text without prior training on the specific categories. Specifically, we use the following models, representing the 5 most-downloaded[6] models in this category:

- **facebook/bart-large-mnli**: A large BART model [32] fine-tuned on the Multi-Genre Natural Language Inference (MNLI) dataset [33].
- **MoritzLaurer/bge-m3-zeroshot-v2.0**: A BGE-M3 model [34] fine-tuned on 33 datasets (5 NLI and 28 non-NLI) covering 389 different classes [35].
- **MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli**: A DeBERTa model [36] fine-tuned on the MNLI [33], FEVER [37], and ANLI [38] datasets.
- **MoritzLaurer/deberta-v3-base-zeroshot-v1.1-all-33**: DeBERTa model [36] fine-tuned on the previously-mentioned 33 datasets [35].
- **mjwong/e5-base-v2-mnli-anli**: An E5 model [39] fine-tuned on the MNLI [33] and ANLI [38] datasets.

[3]https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

[4]https://www.tensorflow.org/text/tutorials/word2vec

[5]https://huggingface.co/

[6]As of March 15th, 2025.

All selected models are transformer-based language models fine-tuned for natural language inference tasks, making them appropriate for zero-shot classification tasks. We evaluate these models on their ability to classify Ansible parameters as either *sensitive* or *non-sensitive* using their specifications and descriptions, by measuring the models' *precision*, *recall*, *accuracy* and *F1-score* on the entire dataset.

2) Fine-tuned LMs: Pretrained models can be further fine-tuned to improve their performance on specific tasks through transfer learning [40]–[42], which trains an existing model on a smaller, task-specific dataset to learn task-specific patterns. This requires fewer samples than training a model from scratch, as the pretrained model already has a general understanding of the language. Fine-tuning is common in NLP and has been shown to significantly improve performance.

Several frameworks exist to fine-tune pretrained models. We use Hugging Face's Transformers[7] as it provides a convenient interface for training and evaluating models, facilitating experimentation with different architectures and hyperparameters.

We apply supervised fine-tuning on the 5 pretrained models selected for the zero-shot classification task. To this end, we provide them the parameters' name, argument specification (type, default value, etc.) and description as input features, and whether the parameter is sensitive as the feature to be predicted. The input features are concatenated into a single string, which is tokenised and converted into a numerical vector representation. Likewise, the feature to be predicted is converted to a binary representation, where *sensitive* parameters are assigned 1, and *non-sensitive* parameters assigned 0. Then, we train the models, adjusting the model's parameters to minimise the loss function. Due to class imbalance (cf. Section III-B), we also experiment with *Random Oversampling* and *Random Undersampling* of the dataset in addition to not applying data balancing. Analogously to the ML models (cf. Section III-B), we use stratified 10-fold cross-validation. We evaluate the fine-tuned models by averaging *precision*, *recall*, *accuracy*, and *F1-score* across all iterations of cross-validation.

D. RQ_3: Comparing Classification Models Against Baselines

To assess the practical benefits of the ML and transformer-based LM classifiers, we compare them against two baselines that identify sensitive parameters using simple string patterns. The first replicates Ansible's "sanity check" for unannotated sensitive parameters, whereas the second replicates the string patterns used to detect hardcoded secrets in GASEL [11], a state-of-the-art security smell detector for Ansible. Both baselines test whether predefined security-related tokens (e.g., "pass" for passwords) appear in parameter names, and apply a filter on the argument type (e.g., boolean values are unlikely to be secret). However, they differ in the concrete string patterns employed. We reimplement both checks within our experimental framework, basing ourselves on the most recent commit in Ansible's codebase and GASEL's replication package.

We apply both baseline implementations to the parameters in our ground truth dataset (cf. Table II) and calculate their *precision*, *recall*, *accuracy*, and *F1-score*. Then, we compare these metrics to those achieved by the best-performing traditional ML (RQ_1) and transformer-based LM (RQ_2) classifiers. Finally, we investigate the overlap between the different approaches to assess whether they discover the same or different sets of sensitive parameters.

E. RQ_4: Classifying Previously-Unseen Parameters

Recall from Section III-A that 14,968 parameters are not included in our ground-truth dataset, as we could not automatically identify whether they are sensitive. They have therefore not been used in the training or evaluation of classifiers and baselines. Our final research question applies the best models from **RQ_1** and **RQ_2**, and the baselines from **RQ_3** on this set of non-annotated parameters. We conduct a manual review of all parameters flagged by any of the approaches, consulting their documentation and the implementation of their respective modules where necessary. This enables us to evaluate the approaches' performance on unseen data, and to potentially uncover previously-unknown sensitive parameters.

IV. Results

This section presents the results for each research question.

A. RQ_1: Accuracy of the Machine Learning Classifiers

Training and evaluation of the various ML configurations described in Section III-B was conducted on a DELL POWEREDGE R730 server, equipped with two INTEL XEON E5-2637 processors, 256 GB of RAM and a 1.6 TB SSD. Due to space constraints, we only discuss the best configuration and refer to our replication package [19] for the remaining results.

COUNTVECTORIZER combined with the feature selection step establishes the best configuration pipeline. Its performance is summarised in Table III for the analysed ML algorithms and data balancing techniques. We observe that the Random Forest classifier without data balancing consistently outperforms the other configurations, achieving the highest *F1-score* (81.7%) and maintaining high *precision* (93.5%), *recall* (72.5%), and *accuracy* (96.7%). This suggests that Random Forest is robust against class imbalance in this specific setup.

We observe that using data balancing improves the *recall* of all algorithms yet substantially decreases their *precision*, overall leading to slightly lower *F1-scores*. Apart from KNN, the choice between oversampling and undersampling seems to have little impact. However, KNN combined with random undersampling achieves a particularly poor *F1-score* of 43% and a *precision* of only 30%, showing its inability to effectively capture the underlying patterns when significant portions of the data are discarded. These results suggest that applying data balancing to our problem domain can result in a more balanced performance, whereas omitting data balancing causes the models to learn more distinctive patterns of sensitive parameters. Such considerations may be important when integrating these approaches into tools, as models trained without data balancing may report fewer false alarms. Conversely, models trained with data balancing

[7]https://huggingface.co/docs/transformers/en/index

TABLE III: Performance comparison of ML algorithms for different balancing techniques. All scores are in percentages.

Model	No Balancing				Random Oversampling				Random Undersampling			
	Prec.	Rec.	F1	Acc.	Prec.	Rec.	F1	Acc.	Prec.	Rec.	F1	Acc.
KNN	95.25	65.06	77.31	96.13	75.42	72.76	74.06	94.83	30.45	77.38	43.70	79.8
Random Forest	**93.57**	**72.65**	**81.79**	**96.72**	82.92	77.47	80.1	96.1	82.83	77.47	80.06	96.09
SVM	94.79	70.3	80.73	96.59	82.42	77.43	79.85	96.04	82.36	77.47	79.82	96.03

TABLE IV: Selected features and mutual information scores.

Feature	Score	Feature	Score
`ansible_password`	0.04301	`axapi`	0.00652
`password`	0.04025	`generated`	0.00652
`authenticating`	0.02522	`azure`	0.00641
`ibm`	0.01916	`mutual`	0.00610
`provided`	0.01615	`based`	0.00606
`classic`	0.01527	`ansible_net_password`	0.00596
`infrastructure`	0.01358	`cred`	0.00581
`access_token`	0.01337	`avi_credentials`	0.00569
`ibmcloud_api_key`	0.01311	`collection_name`	0.00563
`ic_api_key`	0.01311	`aws`	0.00552
`iaas_classic_username`	0.00849	`vmware_password`	0.00518
`iaas_classic_api_key`	0.00846	`token`	0.00518
`softlayer`	0.00845	`log_mode`	0.00515
`argument`	0.00735		

may uncover more sensitive parameters at the expense of more false positive reports. Overall, although data balancing enhances recall and compensates for imbalanced distributions, CountVectorizer combined with feature selection suffices for Random Forest to achieve strong, stable results without further adjustments, making it the most effective and reliable ML configuration in this experimental setting.

Table IV reports the most relevant features extracted during the feature selection step. We can observe terms such as `ansible_password`, `password`, `access_token`, and `ibmcloud_api_key`, which directly relate to credentials and authentication. This suggests that the classifier has successfully learned to associate semantic and lexical patterns that are indicative of sensitive parameters. The appearance of such features with high mutual information scores highlights the model's ability to identify relevant textual features even when such parameters are not explicitly marked as sensitive by developers. Additionally, several features refer to specific cloud providers or infrastructure platforms, such as `ibm`, `azure`, `softlayer`, and `aws`. These tokens likely reflect the application domains in which sensitive parameters are frequently used, e.g. cloud authentication workflows. Moreover, terms like `argument`, `authenticating`, `generated`, and `log_mode` suggest that the model is sensitive not only to explicit secrets but also to contextual evidence about how parameters are used within modules. These features may signal behaviours or conditions under which secrecy is typically required (e.g., authentication steps, logging configurations).

Answer to RQ_1: The combination of CountVectorizer, feature selection, no data balancing and Random Forest achieves the highest *precision* (93.6%) and *F1-score* (81.8%). Additionally, we observe that data balancing increases *recall* but decreases *precision*.

B. RQ_2: Accuracy of the Transformer-based LM Classifiers

The evaluation of the transformer-based language models was performed on a system equipped with an AMD Ryzen 9 7950X 16-core processor, 128 GB of DDR5 RAM, and an NVIDIA GeForce RTX 4090 GPU with 24 GB of VRAM. Table V depicts the results for the zero-shot and fine-tuned models, which we discuss in separate subsections.

1) Zero-shot performance: In the zero-shot setting, the language models achieve high *precision* (84%–89%) and high *recall* (89%–91%). The *F1-scores* (85%–88%) highlight a balanced *precision* and *recall*, while the *accuracy* (89%–91%) shows that the models classify most parameters in the dataset correctly. This indicates that the models are effective at identifying most of the sensitive parameters in the dataset while raising few false alarms. This also suggest that the pretrained models generalise well to the task of detecting sensitive Ansible parameters, even without task-specific training.

The language models outperform the traditional machine learning approaches, which achieved a maximum *F1-score* of 82% (cf. Table III). This demonstrates the effectiveness of their attention mechanism [16] and their extensive pretraining on diverse text corpora [17], which enable capturing complex patterns and relationships in the data without needing task-specific training. This in turn enables them to understand the context and semantics of parameters, leading to improved performance in detecting sensitive Ansible module parameters.

2) Fine-Tuned LLM Performance: Fine-tuning further improves the precision and recall of the pretrained language models. Specifically, most of the fine-tuned models achieve *precision*, *recall*, *F1-score*, and *accuracy* values above 98%. We observe that data balancing has little impact on the fine-tuned models' performance, with random oversampling slightly improving all metrics. The only exception is **bart-large-mnli** fine-tuned with oversampling, whose *precision* and *recall* decreased to 79.95% and 81.77%, respectively, which is lower than the corresponding pretrained model.

Four of the models (**bart-large-mnli**, **DeBERTa-base-mnli**, **deberta-base-zeroshot** and **e5-base-v2-mnli-anli**) achieve a similar performance in the unbalanced fine-tuning setting, with *F1-scores* ranging between 99.76% and 99.80%. Their near-perfect *precision* and *recall* indicate that they produce almost no misclassifications. This shows that fine-tuning enables the models to learn the specifics of the Ansible dataset, improving their performance in identifying sensitive parameters.

Finally, we investigate which tokens carry the most importance in the classification task. To this end, we inspect the tokens with the highest *attention scores*, which are calculated by the language models' attention mechanism [16] to assess the importance of a given token for the task at hand. In our

TABLE V: Performance comparison of zero-shot and fine-tuned transformer-based LMs. All scores are in percentages.

Model	Zero-shot				Fine-tuning (Under)				Fine-tuning (Over)				Fine-tuning (Unbalanced)			
	Prec.	Rec.	F1	Acc.	Prec.	Rec.	F1	Acc.	Prec.	Rec.	F1	Acc.	Prec.	Rec.	F1	Acc.
bart-large-mnli	84.62	89.27	85.81	89.27	99.37	99.33	99.34	99.33	79.95	81.77	80.12	81.77	99.79	99.79	99.79	99.79
bge-m3-zeroshot	87.35	90.13	87.13	90.13	99.19	99.13	99.15	98.13	99.82	99.82	99.82	99.82	97.89	98.80	98.32	98.80
DeBERTa-base-mnli	85.33	89.46	86.15	89.46	99.02	98.93	99.95	98.93	99.81	99.81	99.81	99.81	99.77	99.77	99.77	99.77
deberta-base-zeroshot	88.46	90.43	87.14	90.43	98.95	98.84	98.87	98.84	99.79	99.79	99.79	98.79	99.76	99.76	99.76	98.76
e5-base-v2-mnli-anli	87.51	90.15	86.56	90.15	99.19	99.13	99.14	99.13	99.81	99.81	99.81	98.81	99.80	99.80	99.80	98.80

TABLE VI: Most relevant tokens identified by the attention mechanism of the fine-tuned **deberta-base-zeroshot** model.

Token	Accumulative Att. Score	Token	Accumulative Att. Score
`authentication`	26703.14	`target`	10438.49
`command`	15677.21	`environment`	9863.49
`field`	13139.85	`password`	9644.76
`object`	11843.74	`name`	9075.32
`value`	11123.35	`state`	8604.56

setting, this enables the language models to identify the tokens that are more likely to indicate sensitive parameters, even if the parameter names or descriptions do not follow conventions or are embedded in complex linguistic structures. We calculate the *accumulative attention score* as the sum of the attention scores for each token across all layers of the model. We focus on nouns, as they are more likely to describe parameters in Ansible modules. Table VI depicts the 10 most relevant nouns identified by the attention mechanism of the fine-tuned **deberta-v3-base-zeroshot-v1.1-all-33** model, one of the best performing models (cf. Table V). Several relevant tokens (e.g., `authentication` and `password`) directly relate to authentication, whereas others are generic (e.g., `command`, `value`, etc.) but may be necessary for the language model to understand the context in which the parameters are used.

Answer to RQ$_2$: Pretrained transformer-based language models achieve high *precision* (84–89%) and *recall* (89–91%). Fine-tuning the models enables them to achieve near-perfect results, with *F1-scores* up to 99.8%.

C. RQ$_3$: Comparison of the Approaches

RQ$_1$ and **RQ$_2$** revealed that the **Random Forest without data balancing** and the **fine-tuned deberta-base-zeroshot** classifiers achieve the best performance as traditional ML and transformer-based LM approaches. We compare their results to two baseline approaches, namely Ansible's sanity check and GASEL's string patterns [11], which reported 14,548 and 11,355 sensitive parameters, respectively. Figure 1 compares the *precision*, *recall*, and *F1-score* of the four approaches.

We observe that Ansible's sanity check baseline achieves lower *precision* but relatively high *recall* compared to the GASEL baseline. This highlights a key difference between the two baselines, namely that Ansible's sanity check is aggressive by aiming to identify as many sensitive parameters as possible, whereas GASEL is conservative by attempting to avoid false alarms. The performance of Random Forest is situated between the two baselines, achieving *precision* on-par with GASEL while almost matching the *recall* of Ansible's sanity check.

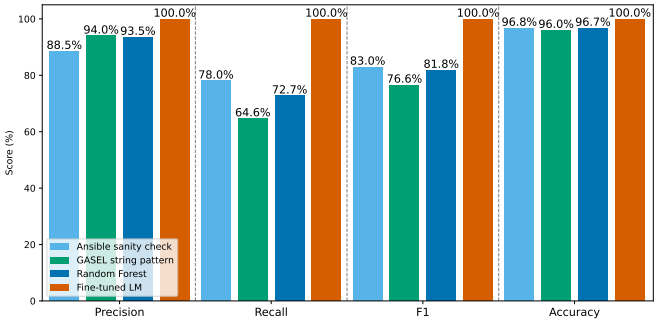

Fig. 1: Comparison of *precision*, *recall*, and *F1-score* for the best-performing ML and LM approaches and baselines.

Fig. 2: Overlap between parameters flagged by the approaches.

The fine-tuned **deberta-base-zeroshot** LM classifier outperforms all other approaches, achieving near-perfect *precision* and *recall* on the ground truth dataset. The high *precision* yet lower *recall* of the other approaches may indicate that many sensitive parameters employ naming patterns or descriptions that are difficult to distinguish from normal parameters without the deep semantic understanding that language models offer.

Figure 2 summarises the number of overlapping reports between the four approaches as a Venn diagram. It shows that there is considerable overlap between the approaches, with many parameters being reported by all approaches together. On the other hand, the large number of reports originating only from the baseline approaches or only from the prediction models suggests that the approaches may be complementary to one another. However, considering the high performance of the language model and the lower precision of the baselines, it

is also possible that the reports that are unique to the baselines are in fact false positives according to the ground truth dataset. This will be investigated in more detail in Section V-B.

Answer to RQ$_3$: The Random Forest model without data balancing achieves performance comparable to the baselines, whereas the fine-tuned language model outperforms all approaches. The approaches may complement one another.

D. RQ$_4$: Classifying Unseen Parameters

After running the baseline approaches and prediction models from **RQ$_3$** on the 14,968 unseen parameters, we obtain 241 unique new reports of potentially sensitive parameters, with some parameters being reported by multiple approaches. Specifically, the Random Forest model flags 224 parameters, while 13, 8, and 4 parameters are reported by the language model, the GASEL baseline, and the Ansible sanity check, respectively. After manually reviewing these reports, we identified only 2 reports as correct, both for parameters partially named `pwd`, with their documentation describing them as passwords. Both parameters were reported by both the GASEL baseline and the LM, but not by the Ansible sanity check nor the Random Forest model. Considering that only 2 of the 14,968 unseen parameters were correctly flagged as secrets, this strongly suggests that the modules in our dataset without `no_log` annotations in fact contain no sensitive parameters.

We also identified several recurring reasons for false positives among the approaches. For instance, all approaches reported parameters that took paths to files containing secrets. Such paths are themselves not secret, yet the approaches reported them as the parameter names did not sufficiently indicate that the parameter expects a path (e.g., a parameter named "`private_key`" instead of "`private_key_path`"). Similarly, the approaches often struggled to distinguish public from private cryptographic keys, with only the latter needing to be secret. This suggests that improving Ansible module parameter naming may further decrease false alarms.

Answer to RQ$_4$: Only 2 new sensitive parameters are detected by both the LM and the conservative baseline, suggesting that Ansible module developers annotate most sensitive parameters correctly as such.

V. Discussion

In this section, we discuss the implications of our findings and potential avenues for future research.

A. Manual Analysis of Misclassifications

To gain qualitative insights into the reason for misclassifications by the classifiers, we conducted a manual investigation of their false positives and negatives against the ground truth.

1) Misclassifications of the ML classifier: Table VII reports some examples of misclassifications by the best-performing ML model, while our replication package [19] contains the results for each single row classified.

We observed that **false negatives** were commonly caused by descriptions that do not include any of the tokens reported in Table IV and extracted during classification. The parameter descriptions may use more generic terms, such as "sensitive information" or "secret text" (cf. Table VII). The absence of more details could lead the classifier to underestimate their secrecy. Furthermore, some descriptions rely solely on vendor-specific acronyms or protocols (e.g., "OSPFv2" or "NXOS") without including generic security keywords. For instance, although the `ospf_auth_text_key` parameter's description contains the term **key** (cf. Table VII), the ML model merged this term with more frequent credential uses (e.g., "`ansible_net_password`") and therefore undervalued the term when it is not connected by contextual tokens like "`access_token`".

TABLE VII: Examples of misclassifications by the Machine Learning model. (**A** = Actual value, **P** = Predicted value)

Param. Name	Parameter Description	A	P
`tls_psk_identity`	*TLS connection uses this PSK identity string. The PSK identity string will be transmitted unencrypted over the network. Therefore, you should not put any sensitive information here. [...]*	T	F
`radkey`	*Authentication key (shared secret text string) for RADIUS clients and servers to exchange. [...] Password that is required for logging on to the server.*	T	F
`ospf_auth_text_key`	*The authentication key for OSPFv2 authentication type text*	T	F
`access_key`	*AWS access key ID. See the AWS documentation for more information about access tokens. [...]*	F	T
`aggregate.sshkey`	*Specifies one or more SSH public key(s) to configure for the given username. This argument accepts a valid SSH key value.*	F	T

As for **false positives**, we observe that a frequent cause of misclassifications arises when the classifier associates the parameter name "`access_key`" (and related AWS-specific tokens) with secret credentials. However, the AWS Access Key ID is in fact a public identifier, while the AWS Secret Key contains the private data. While keywords such as "`access_key`" strongly indicate secrecy to the ML classifier, developers correctly marked it as non-secret. Similar cases can be observed for SSH public keys, which the model considers secret because of keywords such as "SSH" and "key". However, SSH public keys are intended to be shared without exposing sensitive information, while the corresponding private key must remain secret. In summary, the ML model often raises false alarms because it cannot reliably distinguish between public and private credentials.

2) Misclassifications of the language model classifiers: The **deberta-v3-base-zeroshot-v1.1-all-33** classifier reports only 12 misclassifications (7 false positives and 5 false negatives), which are depicted in Table VIII. Only 11 elements are shown as the first two share the same description.

We observe that some parameters, such as the 2nd and 5th rows, were correctly reported by the classifier as secrets, but the developers failed to mark them as such. The model's ability to identify these parameters suggests that it can enhance the security of Ansible modules by detecting secrets that have been overlooked by developers.

TABLE VIII: Misclassifications by the **deberta-v3-base-zeroshot-v1.1-all-33** model. (**A** = Actual, **P** = Predicted)

	Parameter Description	A	P
1	*Name of the key chain.*	F	T
2	*Password that is required for logging on to the server.*	F	T
3	*The username with which to authenticate to the Citrix node.*	F	T
4	*Sendgrid API key to use instead of username/password.*	T	F
5	*Vault token. Token may be specified explicitly.*	F	T
6	*Authentication key for clients and servers to exchange.*	T	F
7	*Secondary password that users might have to provide.*	T	F
8	*Dictionary set by a CyberArk authentication containing the values to perform actions on a logged-on CyberArk session.*	T	F
9	*The source snapshot used to create this disk. You can provide this as a partial or full URL to the resource.*	F	T
10	*Pass credentials to all domains.*	T	F
11	*Specify which groups should have access to what permissions for the storage-system.*	F	T

Conversely, the model predicted some parameters as not secret, while developers marked them as such. Some of these are clearly **false negatives**, such as the 4th, 6th, 7th, and 10th rows in Table VIII. This shows that despite the model's strengths, it may still produce wrong classifications that can cause sensitive parameters to not be properly identified.

B. Qualitative Comparison of the Approaches

To gain qualitative insights into the overlap between the two best classifiers and the baseline approaches, we manually reviewed a random sample for each group of overlapping reports depicted in Fig. 2. We reviewed up to 25 instances per group, i.e., 333 instances in total.

We observed that instances reported only by the ML model or only by either of the baselines are almost always false positives. However, parameters flagged by multiple approaches were often true positives, especially those also reported by the language model. This strongly suggests that the approaches are complementary. Therefore, future work could investigate post-processing an inexpensive baseline's reports with a more costly ML or LM approach to reduce false positives. Alternatively, due to the resource requirements of the prediction models (e.g., models can be multiple gigabytes in size), the prediction models could be used as a final check before releasing an Ansible collection rather than on every code change.

We also encountered some sensitive parameters that were reported by both baselines, but not by either of the prediction models. These belong to a few collections in the dataset in which not all secrets were properly marked. This noise appears to have taught the models, especially the language models, to not treat those parameters in those collections as secret, although they clearly are (e.g., parameters named "password").

Conversely, the language model managed to identify secrets that the baselines missed because the parameter names did not follow their predefined string patterns. For instance, the LM was the only approach to correctly identify a common parameter named "`service_account_contents`" in the *google.cloud* collection as secret, as it may contain private credentials. It also appears that the fine-tuned LM has learned conventions, as some collections mark usernames as secret information, while others do not. The fine-tuned LM manages to distinguish between these, and only suggests usernames as secret in collections that adopt that convention.

Finally, we observe that the baseline approaches frequently misclassified parameters with ambiguous names (e.g., "`key`", "`password_file`"), whereas the prediction models could leverage the description to disambiguate their secrecy. This is especially common for edge cases such as public SSH keys and public AWS access keys (cf. Section V-A1), which the baselines often report as secret. We theorise that general-purpose LLMs (e.g., GPT-4) may be able to better distinguish such cases due to their vast knowledge base and enhanced reasoning capabilities. However, we did not evaluate such LLMs due to their prohibitive costs and the already excellent results of the fine-tuned language model classifiers.

C. Implications of Findings

For practitioners: Our findings uncovered several shortcomings in the baseline approaches. For instance, Ansible's sanity check's inability to discriminate public and private keys leads to many false positives, while GASEL frequently raises false alarms on parameters that take a path to a file containing secrets. Further **improvements to the string patterns** used by these approaches could enhance their capabilities. Moreover, during the manual analyses, we identified multiple sensitive parameters that were not marked with a `no_log` annotation, which can lead to information disclosure vulnerabilities. We therefore implore Ansible module maintainers to **scrutinise the module parameters** to identify missing secret annotations.

For researchers: Our qualitative comparison of the approaches (Section V-B) indicates that the ML and LM classifiers are complementary to the baselines. This further motivates future work to integrate such **prediction models to reduce false alarms raised by inexpensive approaches based on string patterns** [43], [44]. Moreover, we found that fine-tuned LM classifiers excel at distinguishing secret from non-secret data when extensive documentation is present. These insights motivate further research in **applying transformer-based language models for other security-sensitive classification tasks**, such as identifying sources and sinks for taint analysis [15], [45]. However, our experiments also show that fine-tuned language model classifiers may be sensitive to noisy data in the ground truth. Therefore, care must be taken to **use high-quality datasets when fine-tuning LM classifiers**.

VI. Threats to Validity

Multiple factors might have biased the conclusions drawn in our empirical assessment [46]. This section overviews the main threats faced and how they were mitigated, discussing them based on their impact on our study.

a) Internal Validity: The selection of Ansible module parameters considered in our dataset may have an impact on our results. To mitigate this threat, we used a dataset of mature, well-maintained and popular Ansible collections proposed by prior work. Second, our classifiers rely on developer-provided documentation, which may be inaccurate or ambiguous. Although this may cause misclassifications, we argue that the

classifiers need to be able to process suboptimal documentation if applied in practice. Finally, considering parameters that have no `no_log` annotation as implicitly non-secret could cause incorrect entries in the ground truth dataset if developers omitted such annotations. We mitigated this by only considering such parameters if the module contains another parameter with an explicit annotation, suggesting developers scrutinised their modules for sensitive parameters. The results of **RQ$_4$** confirmed that this heuristic was appropriate.

b) Conclusion Validity: The presence of confounding features, i.e., features providing a similar contribution to the performance of the machine learning models, may increase noise when training ML algorithms, biasing its performance [47]. To account for this potential threat, we computed the information gain provided by each feature used to train the models [22]. This allowed us to verify that the tokens extracted in the first step of the machine learning pipeline were orthogonal and contributed individually to the built models.

We did not have a baseline for ML algorithms experimented with, as our work is the first to study ML for sensitive Ansible parameter detection. As such, we experimented with multiple techniques to identify the best algorithm, and compared its result to non-ML baselines. Due to space constraints, we did not discuss all the results in Section III-B. However, our replication package [19] includes all our findings, which researchers can use to understand, reproduce, and extend our ML pipelines to detect sensitive Ansible parameters, or replicate our approach to similar problem domains.

VII. Related Work

A. Infrastructure as Code

Infrastructure as Code is an emerging research domain [48], [49]. Of particular relevance to our work are security smell detection approaches, such as SLIC [4], SLAC [12], GLITCH [50], and GASEL [11], which employ string patterns to identify hardcoded secrets in infrastructure code. Our study improves upon their detection mechanism through ML and LM classifiers.

Other work in IaC has focused on training ML models to identify defects [26], [51] and code smells [52]. More recently, several approaches have been proposed to automatically generate and repair infrastructure code using LLMs [53]–[55]. Most closely related to our work is that of Borovits et al. [56], who identify linguistic inconsistencies between the implementation and documentation of Ansible tasks. Similarly to our work, they leverage textual documentation and a variety ML algorithms for prediction tasks. Our approach differs from this prior work, as our goal is to leverage documentation to identify sensitive parameters rather than implementation inconsistencies, defects, or code generation and repair.

Whereas most research on IaC has focused on the infrastructure code itself, some works instead study the IaC tool implementations. Prior work has studied testing practices for Ansible modules [57], defects in IaC tool implementations [20], [58], and software supply chains backing IaC tools [21]. Our work is complementary to these studies, as we aim to identify sensitive information within the tool implementation.

B. Predicting Sensitive Information

Due to the dangers associated with embedding secret information in source code, a large body of research has focused on identifying such *hardcoded secrets*. Early detection approaches identified potential secrets based on the entropy of hardcoded text, using regular expressions for known structured secrets (e.g., AWS keys), or using regular expressions on variable identifiers [59], [60]. However, as this generates many false positives, recent approaches reduce such false reports using ML models [43], [61], [62] or LLMs [44]. Our approach differs in that we do not aim to detect the secrets themselves but rather the parameters that accept them.

Therefore, more closely related to our work are approaches that identify security-relevant methods, such as sources and sinks used in taint analysis. Several such approaches leverage ML models using method names and signatures [14], [15], [63]–[67]. However, to the best of our knowledge, the only approach that also uses documentation is DocFlow [45], which uses sentence embeddings and prediction models to identify taint sources and sinks in Android APIs. Contrary to our work, they do not assess pretrained LM classifiers.

VIII. Conclusion

This paper presented an assessment of Machine Learning and pretrained transformer-based Language Model classifiers to identify sensitive parameters in Ansible modules. We collected a ground-truth dataset of over 160,000 parameters of well-maintained Ansible modules, of which more than 16,000 accept secret values. Based on this dataset, we trained a variety of ML models under various data balancing settings, as well as pretrained zero-shot and fine-tuned LM classifiers. We compared the best of these models against two baseline approaches that use string patterns.

Our findings show that the best ML model, Random Forest without data balancing, achieves a performance comparable to the two baselines, achieving a higher *precision* (93.6%) than an aggressive baseline and reaching higher *recall* (72.7%) than a conservative baseline. The zero-shot pretrained LM classifiers achieve a similar *precision* to the other approaches (84.6–88.5%), while reaching a considerably higher *recall* (89.2–90.4%). Finally, fine-tuning the pretrained language models enables them to achieve near-perfect *precision* and *recall*.

A qualitative comparison of the approaches suggests that they complement one another, with both the LM and the baselines identifying sensitive parameters that the others cannot. This motivates further study of language model classifiers to predict security-relevant code elements based on their documentation, as well as integrating prediction models to reduce false positives caused by inexpensive yet inaccurate approaches based on string patterns.

Acknowledgements

This research was partially funded by the "Cybersecurity Research Programme Flanders" project and by the Fonds voor Wetenschappelijk Onderzoek Vlaanderen (FWO) Strategic Basic Research (SBO) BaseCamp Zero under Project S000323N.

References

[1] K. Morris, *Infrastructure as Code: Managing Servers in the Cloud*, 1st ed. O'Reilly, 2016.

[2] StackExchange, Inc. (2023) 2023 annual StackOverflow developer survey. [Online]. Available: https://survey.stackoverflow.co/2023/#section-most-popular-technologies-other-tools

[3] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, "Adoption, support, and challenges of infrastructure-as-code: Insights from industry," in *Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution, Industrial Track*, 2019, pp. 580–589.

[4] A. Rahman, C. Parnin, and L. A. Williams, "The seven sins: security smells in infrastructure as code scripts," in *Proceedings of the 41st International Conference on Software Engineering (ICSE 2019)*. IEEE / ACM, 2019, pp. 164–175.

[5] CVE-2020-1716. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2020-1716

[6] CVE-2019-10217. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2019-10217

[7] CVE-2020-25646. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2020-25646

[8] CVE-2021-3447. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2021-3447

[9] CVE-2021-20180. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2021-20180

[10] CVE-2021-20191. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2021-20191

[11] R. Opdebeeck, A. Zerouali, and C. D. Roover, "Control and data flow in security smell detection for infrastructure as code: Is it worth the effort?" in *20th IEEE/ACM International Conference on Mining Software Repositories (MSR 2023)*. IEEE, 2023, pp. 534–545.

[12] A. Rahman, M. R. Rahman, C. Parnin, and L. A. Williams, "Security smells in ansible and chef scripts: A replication study," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 1, pp. 3:1–3:31, 2021.

[13] N. Saavedra and J. F. Ferreira, "GLITCH: automated polyglot security smell detection in infrastructure as code," in *37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022)*. ACM, 2022, pp. 47:1–47:12.

[14] W. M. Rodrigues, F. N. Walmsley, G. D. C. Cavalcanti, and R. M. O. Cruz, "Security relevant methods of Android's API classification: A machine learning empirical evaluation," *IEEE Transactions on Computers*, vol. 72, no. 11, pp. 3273–3285, 2023.

[15] G. Piskachev, L. N. Q. Do, and E. Bodden, "Codebase-adaptive detection of security-relevant methods," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2019, pp. 181–191.

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention Is All You Need," *Advances in neural information processing systems*, vol. 30, 2017.

[17] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 2019, pp. 4171–4186.

[18] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language Models are Few-Shot Learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[19] R. Opdebeeck, V. Pontillo, C. Velzquez-Rodrguez, W. De Meuter, and C. De Roover. Replication package for Ansible sensitive parameter prediction study. [Online]. Available: https://doi.org/10.6084/m9.figshare.29269091

[20] G.-P. Drosos, T. Sotiropoulos, G. Alexopoulos, D. Mitropoulos, and Z. Su, "When your infrastructure is a buggy program: Understanding faults in infrastructure as code ecosystems," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, p. 359, 2024.

[21] R. Opdebeeck, B. Adams, and C. De Roover, "Analysing software supply chains of infrastructure as code: Extraction of Ansible plugin dependencies," in *Proceedings-2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2025.

[22] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[23] G. E. Batista, R. C. Prati, and M. C. Monard, "A study of the behavior of several methods for balancing machine learning training data," *ACM SIGKDD explorations newsletter*, vol. 6, no. 1, pp. 20–29, 2004.

[24] M. Stone, "Cross-validatory choice and assessment of statistical predictions," *Journal of the royal statistical society: Series B (Methodological)*, vol. 36, no. 2, pp. 111–133, 1974.

[25] G. Quattrocchi and D. A. Tamburri, "Predictive maintenance of infrastructure code using fluid datasets: An exploratory study on Ansible defect proneness," *Journal of Software: Evolution and Process*, vol. 34, no. 11, p. e2480, 2022.

[26] S. Dalla Palma, D. Di Nucci, F. Palomba, and D. A. Tamburri, "Within-project defect prediction of infrastructure-as-code using product and process metrics," *IEEE Transactions on Software Engineering*, vol. 48, no. 6, pp. 2086–2104, 2021.

[27] E. Van der Bent, J. Hage, J. Visser, and G. Gousios, "How good is your Puppet? an empirically defined and validated quality model for Puppet," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 164–174.

[28] A. Rahman and L. Williams, "Characterizing defective configuration scripts used for continuous deployment," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 34–45.

[29] ——, "Source code properties of defective infrastructure as code scripts," *Information and Software Technology*, vol. 112, pp. 148–163, 2019.

[30] W. S. Noble, "What is a support vector machine?" *Nature biotechnology*, vol. 24, no. 12, pp. 1565–1567, 2006.

[31] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[32] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension," 2019. [Online]. Available: https://arxiv.org/abs/1910.13461

[33] A. Williams, N. Nangia, and S. R. Bowman, "A broad-coverage challenge corpus for sentence understanding through inference," 2018. [Online]. Available: https://arxiv.org/abs/1704.05426

[34] J. Chen, S. Xiao, P. Zhang, K. Luo, D. Lian, and Z. Liu, "BGE M3-Embedding: Multi-Lingual, Multi-Functionality, Multi-Granularity Text Embeddings Through Self-Knowledge Distillation," 2024.

[35] M. Laurer, W. van Atteveldt, A. Casas, and K. Welbers, "Building Efficient Universal Classifiers with Natural Language Inference," Dec. 2023, arXiv:2312.17543 [cs]. [Online]. Available: http://arxiv.org/abs/2312.17543

[36] P. He, X. Liu, J. Gao, and W. Chen, "DeBERTa: Decoding-Enhanced BERT with Disentangled Attention," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=XPZIaotutsD

[37] J. Thorne, A. Vlachos, C. Christodoulopoulos, and A. Mittal, "FEVER: a Large-scale Dataset for Fact Extraction and VERification," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. New Orleans, Louisiana: Association for Computational Linguistics, Jun. 2018, pp. 809–819. [Online]. Available: https://aclanthology.org/N18-1074/

[38] Y. Nie, A. Williams, E. Dinan, M. Bansal, J. Weston, and D. Kiela, "Adversarial NLI: A New Benchmark for Natural Language Understanding," *arXiv preprint arXiv:1910.14599*, 2019.

[39] L. Wang, N. Yang, X. Huang, B. Jiao, L. Yang, D. Jiang, R. Majumder, and F. Wei, "Text embeddings by weakly-supervised contrastive pre-training," *arXiv preprint arXiv:2212.03533*, 2022.

[40] K. Weiss, T. M. Khoshgoftaar, and D. Wang, "A survey of transfer learning," *Journal of Big data*, vol. 3, pp. 1–40, 2016.

[41] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" *Advances in Neural Information Processing Systems*, vol. 27, pp. 3320–3328, 2014. [Online]. Available: https://papers.nips.cc/paper/5347-how-transferable-are-features-in-deep-neural-networks

[42] Y. Ganin, E. Ustinova, H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, M. Marchand, and V. Lempitsky, "Domain-adversarial training of neural networks," *Journal of Machine Learning Research*, vol. 17, no. 59, pp. 1–35, 2016. [Online]. Available: http://jmlr.org/papers/v17/15-239.html

[43] A. Saha, T. Denning, V. Srikumar, and S. K. Kasera, "Secrets in source code: Reducing false positives using machine learning," in *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*, 2020, pp. 168–175.

[44] M. N. Rahman, S. Ahmed, Z. Wahab, S. M. Sohan, and R. Shahriyar, "Secret breach detection in source code with large language models," 2025. [Online]. Available: https://arxiv.org/abs/2504.18784

[45] M. Tileria, J. Blasco, and S. K. Dash, "DocFlow: Extracting taint specifications from software documentation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. New York, NY, USA: ACM, 2024.

[46] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer, 2012.

[47] R. M. Obrien, "A caution regarding rules of thumb for variance inflation factors," *Quality & quantity*, vol. 41, no. 5, pp. 673–690, 2007.

[48] A. Rahman, R. Mahdavi-Hezaveh, and L. A. Williams, "A systematic mapping study of infrastructure as code research," *Inf. Softw. Technol.*, vol. 108, pp. 65–77, Apr. 2019.

[49] M. Chiari, M. De Pascalis, and M. Pradella, "Static analysis of infrastructure as code: a survey," in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2022, pp. 218–225.

[50] N. Saavedra and J. F. Ferreira, "GLITCH: Automated polyglot security smell detection in infrastructure as code," in *37th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2022.

[51] M. Begoug, M. Chouchen, A. Ouni, E. Abdullah Alomar, and M. W. Mkaouer, "Fine-grained just-in-time defect prediction at the block level in infrastructure-as-code (IaC)," in *Proceedings of the 21st International Conference on Mining Software Repositories*. ACM, 2024, pp. 100–112.

[52] S. Dalla Palma, C. van Asseldonk, G. Catolino, D. Di Nucci, F. Palomba, and D. A. Tamburri, ""Through the looking-glass ..." An empirical study on blob infrastructure blueprints in the topology and orchestration specification for cloud applications," *Journal of Software: Evolution and Process*, vol. 36, no. 4, Apr. 2024.

[53] P. Sahoo, S. Pujar, G. Nalawade, R. Genhardt, L. Mandel, and L. Buratti, "Ansible Lightspeed: A code generation service for IT automation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2024, pp. 2148–2158.

[54] B. Darnell, H. Chopra, A. Councilman, D. Grove, Y.-X. Wang, and V. Adve, "An empirical comparison of code generation approaches for Ansible," in *Proceedings of the ACM/IEEE 2nd International Workshop on Interpretability, Robustness, and Benchmarking in Neural Software Engineering*. New York, NY, USA: ACM, 2024, pp. 1–6.

[55] J. Diaz-De-Arcaya, J. López-De-Armentia, G. Zárate, and A. I. Torre-Bastida, "Towards the self-healing of infrastructure as code projects using constrained LLM technologies," in *Proceedings of the 5th ACM/IEEE International Workshop on Automated Program Repair*. New York, NY, USA: ACM, 2024, pp. 22–25.

[56] N. Borovits, I. Kumara, D. Di Nucci, P. Krishnan, S. Dalla Palma, F. Palomba, D. A. Tamburri, and W.-J. v. d. Heuvel, "FindICI: Using machine learning to detect linguistic inconsistencies between code and natural language descriptions in infrastructure-as-code," *Empirical Software Engineering*, vol. 27, no. 7, pp. 1–30, 2022.

[57] M. M. Hasan, F. A. Bhuiyan, and A. Rahman, "Testing practices for infrastructure as code," in *Proceedings of the 1st ACM SIGSOFT International Workshop on Languages and Tools for Next-Generation Testing*. ACM, 2020, pp. 7–12.

[58] M. M. Hassan, J. Salvador, S. K. K. Santu, and A. Rahman, "State reconciliation defects in infrastructure as code," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1865–1888, Jul. 2024.

[59] V. S. Sinha, D. Saha, P. Dhoolia, R. Padhye, and S. Mani, "Detecting and mitigating secret-key leaks in source code repositories," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 396–400.

[60] S. K. Basak, J. Cox, B. Reaves, and L. Williams, "A comparative study of software secrets reporting by secret detection tools," in *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2023, pp. 1–12.

[61] J. Zhou, Z. Zhang, L. Ying, H. Chai, J. Cao, and H. Duan, "Hey, your secrets leaked! detecting and characterizing secret leakage in the wild," in *2025 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE, May 2025, pp. 449–467.

[62] R. Feng, Z. Yan, S. Peng, and Y. Zhang, "Automated detection of password leakage from public github repositories," in *Proceedings of the 44th International Conference on Software Engineering*. New York, NY, USA: ACM, 2022, pp. 175–186.

[63] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

[64] D. Sas, M. Bessi, and F. Arcelli Fontana, "Automatic detection of sources and sinks in arbitrary java libraries," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2018, pp. 103–112.

[65] X. Zhang, J. Heaps, R. Slavin, J. Niu, T. Breaux, and X. Wang, "DAISY: Dynamic-analysis-induced source discovery for sensitive data," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, May 2023.

[66] P. Bian, B. Liang, J. Huang, W. Shi, X. Wang, and J. Zhang, "SinkFinder: harvesting hundreds of unknown interesting function pairs with just one seed," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, 2020, pp. 1101–1113.

[67] A. L. S. Pupo, J. Nicolay, K. Efthymiadis, A. Nowé, C. D. Roover, and E. G. Boix, "GUARDIAML: machine learning-assisted dynamic information flow control," in *26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2019)*. IEEE, 2019, pp. 624–628.