Detecting Backdoors in Collaboration Graphs of Software Repositories

Tom Ganz
tom.ganz@sap.com
SAP SE
Germany

Inaam Ashraf
mashraf@techfak.uni-bielefeld.de
Bielefeld University
Germany

Martin Härterich
martin.haerterich@sap.com
SAP SE
Germany

Konrad Rieck
rieck@tu-berlin.de
Technische Universität Berlin
Germany

ABSTRACT

Software backdoors pose a major threat to the security of computer systems. Minor modifications to a program are often sufficient to undermine security mechanisms and enable unauthorized access to a system. The direct approach of detecting backdoors using static or dynamic program analysis is a daunting task that becomes increasingly futile with the attacker's capabilities. As a remedy, we introduce an orthogonal strategy for the detection of software backdoors. Instead of searching for concealed functionality in program code, we propose to analyze how a software has been developed and locate clues for malicious activities in its version history, such as in a Git repository. To this end, we model the version history as a collaboration graph that reflects *how*, *when* and *where* developers have committed changes to the software. We develop a method for anomaly detection using graph neural networks that builds on this representation and is able to detect spatial and temporal anomalies in the development process. We evaluate our approach using a collection of real-world backdoors added to Github repositories. Compared to previous work, our method identifies a significantly larger number of backdoors with a low false-positive rate. While our approach cannot rule out the presence of software backdoors, it provides an alternative detection strategy that complements existing work focused only on program analysis.

CCS CONCEPTS

• **Security and privacy → Software and application security**; **Software security engineering**; • **Computing methodologies → Machine learning**.

KEYWORDS

Software Repositories, Neural Networks, Anomaly Detection

ACM Reference Format:
Tom Ganz, Inaam Ashraf, Martin Härterich, and Konrad Rieck. 2023. Detecting Backdoors in Collaboration Graphs of Software Repositories. In *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy (CODASPY '23), April 24–26, 2023, Charlotte, NC, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3577923.3583657

CODASPY '23, April 24–26, 2023, Charlotte, NC

ACM ISBN 979-8-4007-0067-5/23/04.
https://doi.org/10.1145/3577923.3583657

1 INTRODUCTION

Backdoors are a notorious security threat in software development. Minor tweaks to an authentication or encryption routine are often enough to lift protection mechanisms and provide an attacker with access to sensitive data. In particular, in open-source software projects with hundreds of developers, it is hard to spot such manipulations and differentiate legitimate fixes from carefully crafted backdoors. While techniques for program analysis and code isolation might potentially unveil backdoors [e.g., 32, 35, 37], they are hindered by the complexity of code and the inherent limitations of automatic program analysis [26]. Consequently, there exist numerous cases where developers unnoticeably sneaked backdoor functionality into open-source projects and caused harm to their users [23, 41]. As an example, a malicious patch was inserted into the Git repository of PHP in 2021. The patch was disguised as a typo fix, but actually contained a backdoor that compromised the general security of the web platform [13].

Fortunately, however, in modern software development, code changes rarely go unobserved. Instead, version control systems, such as Git and SVN, meticulously track any change and contributor to a software repository, providing a clear view of the collaborative development effort. This view not only helps locate software bugs and vulnerabilities but also provides valuable clues for identifying anomalous code changes. These clues can manifest in unusual commit messages, code locations or even development times. Unlike program code, the information in a version control system is less obstructed by technical complexity and can therefore serve as an alternative source for spotting malicious activities.

In this paper, we thus focus on detecting backdoors in software through anomalies in the development process. To this end, we model the version history of software as a collaboration graph that reflects *how*, *when* and *where* developers have committed changes to it. Based on this compact representation, we introduce a method for anomaly detection that is capable of identifying spatial and temporal anomalies. Technically, this method is based on a combination of graph neural networks and one-class learning, where the neural network learns a representation of the collaboration graph, while the one-class learner spots unusual activities in it. Compared to

prior work, this learning-based approach spares us from manually defining how a backdoor manifests in the version history and thus provides a general detection strategy.

We investigate the detection performance of our approach in an evaluation with 109 Github repositories, covering a total of 100.000 commits (80k for training and 20k for testing). As detection targets, we consider real backdoors that have been previously found in Github repositories and investigate how they can be identified solely through commits in the version history. As part of this evaluation, we compare our approach to state-of-the-art methods for detecting malicious and anomalous commits. We find that our approach based on collaboration graphs outperforms these baselines methods by detecting three times more backdoors while also reducing the false-positive rate. In summary, our method identifies 78.9% of the backdoors in the software repositories with 0.7% false alarms on average.

Despite any research effort, the detection of backdoors remains a problem that can never be solved completely [see 26]. Still, we argue that our approach significantly raises the bar for adversaries to introduce backdoors during software development. In conjunction with other techniques for mitigating backdoor functionality, such as static program analysis and code isolation, our work strengthens the protection of software from silent code manipulations.

Contributions. We make the following contributions in this work:

- *Backdoors as collaboration anomalies.* We model the problem of locating backdoors in software as anomaly detection in collaboration graphs, complementing existing work based on static program analysis.
- *Detection using graph neural networks.* We propose a novel graph-based detection method that infers a model of normal collaboration in a software repository and identifies backdoors as deviations thereof.
- *Empirical evaluation on real backdoors.* We introduce a dataset with and without backdoors of 109 repositories resulting in a graph model with over 100k commits and more than a million nodes. On this data, our approach significantly outperforms related methods for backdoor detection.

The rest of this paper is organized as follows: We describe the challenges of backdoor detection in Section 2. Section 3 then introduces our approach for identifying backdoors in collaboration graphs. Our experimental setup and the evaluation of our approach are presented in Section 4 and Section 5, respectively. We discuss related work in Section 6 and conclude the paper in Section 7.

2 PROBLEM SETTING

The detection of backdoors in software is an extremely challenging problem. The attacker can take great care to cloak malicious functionality and evade detection. For example, a backdoor might be disguised as an inconspicuous fix or a minor code improvement. To detect such modifications in a software repository, the examining developer needs to have fundamental knowledge of the project. If the open-source project is large, with over hundreds of developers, this knowledge is often split among the contributors, as they are more specialized in smaller areas within the project. Conversely, if the project is small, the effort of cross-reviewing every commit quickly becomes too costly. As a result, the necessary knowledge for spotting sneaked-in backdoor code is usually not available when a developer reviews a code change.

Static and dynamic program analysis provides means to detect software bugs and vulnerabilities, mitigating this situation [e.g., 2, 30, 42, 44]. However, these approaches are unlikely to spot backdoor code in commits, as they are designed to detect unintentional defects rather than intentional tampering. Similarly, approaches to isolate code functionality can help reduce the attack surface for backdoors. Yet, these techniques are not always applicable and require a considerable engineering effort for integration [32, 37]. Motivated by these difficulties, we tackle the problem of backdoor detection from a different perspective. In particular, we expect a malicious change to a project to be detectable as an *anomaly* in the development process. That is, we aim at spotting irregularities in the characteristics of the change rather than the resulting code in the software repository

Let us, as an example, look at the backdoor that was introduced to the PHP platform. To carry out the attack, the adversary impersonated the project maintainer[1]. The malicious commit was disguised as a harmless typo fix. As seen in Listing 1, it introduces a backdoor that is triggered as soon as the user-agent header evaluates to `zerodium`. It then becomes possible to execute arbitrary code placed directly after this string.

```
static void php_zlib_output_compression_start(void)
{
  zval zoh;
  php_output_handler *h;
  zval *enc;
    ...
    if (strstr(Z_STRVAL_P(enc), "zerodium")) {
      zend_try {
        zend_eval_string(Z_STRVAL_P(enc)+8, NULL,
        "REMOVETHIS: sold to zerodium, mid 2017");
  ...
```

Figure 1: Malicious commit from PHP disguised as "typo fix". Malicious changes are highlighted in red.

Without knowledge of the underlying context, neither a static analyzer nor a vulnerability detection tool would consider this piece of code insecure. While cryptographic signatures can protect from impersonations in Git, they are often omitted and sometimes even forged. In this particular case, the developer's signature key was compromised, rendering the detection of the impersonation impossible. In contrast, the operation of the malicious commit contains relevant clues that deviate from the usual development. For instance, the malicious commit was introduced at around 06:00 AM. However, the past 20 commits to the master branch by the original contributor were all issued later than 06:00 PM, suggesting that the commit deviates from his usual working hours. Consequently, we motivate that the analysis of the development process provides an alternative source for detecting software backdoors.

2.1 Challenges

While it may seem straightforward to investigate the version history of a software project, there exist several challenges that hinder such

[1] https://github.com/php/php-src/commit/c730aa2

an analysis. In this section we describe the challenges which we effectively address in this paper. Later, in Section 3, we explain how our method overcomes the underlying problems.

Complex relations. A commit can be thought of as a transition from one state of the project to the next. In the case of the PHP backdoor, the attack was conducted over two subsequent commits. An analysis approach focusing on individual commits would overlook such patterns. Hence, the approach needs to inspect a commit together with its preceding and subsequent commits. Furthermore, several properties of the code change need to be considered as well, such as collaborations between contributors, temporal relationships, project files belonging to or changed by the same contributors and so on. All these properties can play a crucial role in detecting injected backdoors in commits.

Large variance. Current works by Gonzalez et al. [12] and Goyal et al. [14] focus on manually engineered detection rules. However, these patterns of misuse are limited by the prior knowledge of the practitioners who produce them. Since anomalous commits may exhibit a large variance in their behavior and appearance, a fixed rule set is not able to capture all possible sources of anomalous properties. Moreover, the statistical properties of the contributors' behavior stem from vastly different underlying distributions, rendering manual modeling of regular behavior intractable.

Semantic reasoning. The PHP backdoor from Listing 1 presents a commit titled *typo fix* that modifies two lines in the source code of a C function. The fact that a fix for a typo causes more than one change is suspicious and a potential indicator of anomalous activity. As a result, a detection approach needs to be able to detect semantic discrepancies between what the developer claims to contribute and what she actually commits to the repository. This is a challenging task, as it requires providing a link between natural language text and code changes.

3 METHODOLOGY

Equipped with an understanding of the key challenges for detecting backdoors in software development, we are ready to introduce our method. In the following, we describe its four main components and how they act together. First, we present the concept of *collaboration graphs* that serves as a basis for modeling the development process (Section 3.1). Next, we derive our learning-based detection approach that embeds individual *features* in a vector space (Section 3.2), constructs an *embedding* of the entire graph using graph-neural networks (Section 3.3) and finally employs a one-class learner for *anomaly detection of nodes* (Section 3.4).

3.1 Collaboration Graphs

Version control systems, such as Git and SVN, have become ubiquitous in software development, as they make collaborative development in a team transparent. Metadata extracted from such a system intrinsically yields a representation as a graph that reveals collaborations between authors, changes to code, and connections to past, present and future states [33]. We represent these relations as a directed graph $G(V, E)$ composed of nodes V and directed edges $E \subseteq V \times V$. Sticking to the taxonomy established by *Git*, the central node type within this graph is a *commit*. We enrich this representation by introducing node types for *branches*, *files*, *developers*, and *methods* (functions) as shown in Figure 2. As a result, we obtain a heterogeneous graph that characterizes the development process within a software repository in a compact form.

In particular, we model the relation between commits by drawing a directed edge from one commit to its successor. The relation between a developer who changes a project state and the new commit which is thereby created is modeled as a directed edge from the developer to the commit. Also, the files and methods that have been modified by a commit are modeled as nodes while having a relationship as depicted in Figure 2. Thus, a commit points to its changed method and file nodes. The file node points to its containing method nodes. Further, we add a directed edge from the project's branch to all its underlying commits.

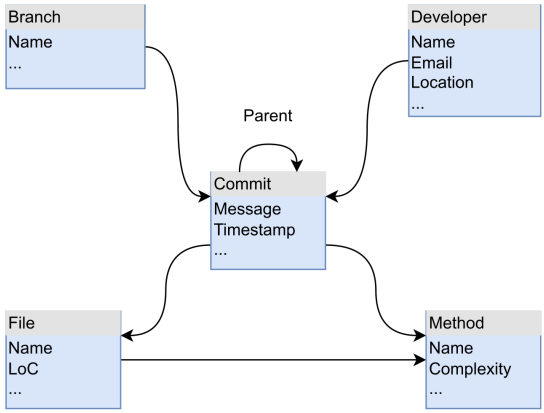

Figure 2: Model of a collaboration graph.

Collaboration graphs of a software repository exhibit rich properties about the project and the development process. For example, the graph captures statistics of the developers' behavior, such as their working hours, recent collaborations among developers, the change frequency per file or interconnections between commonly changed files. Besides the topological structure of the collaboration graph, we augment it with further attributes.

- *Developer node:* We enrich the node of a developer with statistical information about her behavior, including the overall number of commits and projects, the account age, name, email, number of merge and non-merge commits and if available, her location.
- *Commit node:* We attach additional attributes to the commit node, such as the commit message, the timestamp, merge type and whether the commit is signed off or not. Moreover, we add code-derived metrics taken from the delta maintainability model (DMM), such as the unit value of the interfacing property, the size property and the complexity property [4].
- *File and method node:* We attach the name and mime-type to all file nodes. Moreover, we add the number of past and present methods and the lines of code to these nodes. Finally, for the method node, we only include the parent file name and its method name.

Furthermore, we attach several attributes to the edges of the graph. We add the commit timestamp to the edges pointing to the descendant file and method nodes. We add the type of file update to the edge connecting the commit node to its file nodes, i.e., whether it was modified, created or deleted. Finally, we add the fan-in and fan-out of the method (input and output degree), the start and end lines, the token count and the cyclomatic complexity to all edges connecting a file to method nodes

Overall, the proposed augmentation of the collaboration graph provides us with a comprehensive view of the information contained in a software repository. By modeling different relations as directed edges, we are able to track changes in the project state across commits, developers, files, methods and even time — establishing a rich foundation for anomaly detection.

3.2 Feature Representation

Several of the defined attributes are numeric in nature and do not need any pre-processing to be accessible by machine-learning techniques. Non-numeric attributes, however, need to be embedded in a vector space, so that they become usable by learning models.

Identifiers. We encode all identifiers numerically as unique labels, such as branch names, project identifiers, developer names, email addresses, locations, file names, file types, and method names. In addition, we encode the Levenshtein distance between the developer names and email addresses as well as between the file names and method names. These additional numerical values enable our detection method to spot inconsistencies in combinations of these strings, such as typosquatting or fake addresses.

Natural text. For longer texts like commit messages, we use a *word2vec* embedding based on the implementation provided by Gensim [10]. In particular, we make use of the continuous bag-of-words (CBOW) algorithm using negative sampling. We employ a standard configuration of the algorithm by setting the window size to 5 and the dimension to 100. Since we expect the distribution of words to differ per repository, we train an embedding for each collaboration graph.

Time information. We decompose a commit's Unix timestamp in the pre-processing step to be more insightful for our model. To differentiate between commits made on weekdays and weekends, we use the one-hot encoding of the *day of the week*. Similarly, we one-hot encode the *month of the year* to get 12 features. To let the model learn the sequential nature of the commits, we use the following transformation:

(1) Starting from a fixed reference date (January 1, 2005), we compute the number of days from this reference to the commit, which gives us a continuous and increasing numeric *day* feature.
(2) In addition, we encode hour h, minute m and second s of the day by projecting it via

$$\text{time} = \frac{1}{2}\Big(1 - \cos\big(2\pi \cdot (h + (m + s/60)/60)/24\big)\Big)\,, \tag{1}$$

This gives us a periodic feature that starts from 0 at the start of the day, goes to 1 at midday and finally drops back to zero at the end of the day.

3.3 Embedding Collaboration Graphs

For anomaly detection in collaboration graphs, we use the recent learning model of *graph neural networks* (GNNs) with convolutional layers. The success of convolutional neural networks in vision is attributed to their use of filters to successively aggregate information from small regions of an image and learn to represent elementary shapes, textures and complex features. Similarly, GNNs aim to solve learning tasks on graphs by exploiting the relations of nodes and edges in different proximity. Hence, we use convolutional layers to generate representations that capture the graph structure as well as the feature representations. In particular, we consider one vector space of dimension d for each node and one of dimension k for each edge. That is, we define $X = h_v^{(0)} \in \mathbb{R}^{|V| \times d}$ as the *node features* and likewise $F = h_e^{(0)} \in \mathbb{R}^{|E| \times k}$ as the *edge features*.

In general, graph convolutional networks define a message-passing algorithm which for each layer $l = 0, 1, 2, \ldots$ comprises three learnable differentiable functions [16, 39, 43]:

(1) The function ρ constructs a message m_{vu} from the current node features h_v, the node features h_u from the neighboring nodes $\mathcal{N}(v)$ and edge features $h_{e_{vu}}$ of the edges pointing to the current node from its neighbors.

$$m_{vu}^{(l)} = \rho^{(l)}(h_v^{(l)}, h_u^{(l)}, h_{e_{vu}}^{(l)}) \text{ for } u \in \mathcal{N}(v) \tag{2}$$

(2) The aggregation function ζ takes an input $\{m_{vu}^{(l)} | u \in \mathcal{N}(v)\}$, where $m_{vu} \in \mathbb{R}^d$, is an unordered message tuple. It then generates an aggregated message $m_v \in \mathbb{R}^d$.

$$m_v^{(l)} = \zeta^{(l)}(m_{vu}^{(l)} | u \in \mathcal{N}(v)) \tag{3}$$

To be well defined ζ must be permutation invariant. Hence, popular aggregation functions, for instance, include the mean, max and sum functions [16, 19, 43].
(3) The function ϕ combines the current node features h_v with the aggregated message m_v to finally generate the updated node features.

$$h_v^{(l+1)} = \phi^{(l)}(h_v^{(l)}, m_v^{(l)}) \tag{4}$$

Recent research has indicated that different generalized aggregation functions work better for different tasks. We build on the aggregation layers *GENConv* introduced by Li et al. [20] that improve the performance on diverse GCN tasks and scale well with large graph topologies.

Based on these layers, we construct a graph-variational autoencoder (GVAE) similar to Kingma and Welling [18]. This GVAE consists of an encoder and decoder model. While the former embeds the input in a latent space using a Gaussian distribution model parameterized by mean μ and standard deviation σ, the latter reconstructs the embedded input from a latent Gaussian distribution:

$$p(z|h^{(L)}) = \mathcal{N}(z|\mu, \sigma^2) \rightsquigarrow Z = \mu + \sigma\epsilon,\ \epsilon \sim \mathcal{N}(\epsilon|0, 1) \tag{5}$$

Here $h^{(L)}$ is the latent embedding generated by the encoder as in Equation (4) for the final layer L. When training a variational autoencoder, the objective function not only minimizes the reconstruction loss but also maximizes the likelihood of the data by minimizing the Kullback-Leibler divergence between the true latent posterior distribution $p(z|x)$ and the estimated latent posterior

distribution $q(z|x)$. The combined loss is given by

$$\mathcal{L} = \mathcal{L}_{reconst} + \mathcal{L}_{kl}. \tag{6}$$

In our setting, the GVAE is trained to reconstruct node and edge features, while respecting the discrete topology of the graph. Since the collaboration graph is heterogeneous and contains different node types, all node features are first zero-padded to a common feature dimension d' and then passed through a single feed-forward network, resulting in a new homogeneous node feature vector of size $d = 128$. Similarly, we pass the zero-padded edge features with dimension k' through a single feed-forward network to obtain a new edge feature vector of size $k = 128$. Since in the GNN context, features are expected to have the same meaning in all feature dimensions, zero-padding alone is generally discouraged. As a remedy, we project all node and edge features onto a common lower-dimensional space [17].

In summary, the GVAE creates a latent representation characterizing the entire structure of the graph that later serves as the basis for modeling normality and spotting unusual activities.

3.4 Anomaly Detection on Nodes

We are finally ready to use the encoder of the GVAE to analyze and detect anomalies in collaboration graphs. In fact, it enables us to obtain embeddings for all nodes of the collaboration graphs and to apply a one-class learner just as if the data were Euclidean. Before presenting our semi-supervised learner in Section 3.4, we start with a simpler unsupervised version of it.

One-class learning. There exists a large variety of approaches for anomaly detection. For our approach, we focus on a recent model called *Deep Support Vector Data Description* (Deep SVDD) proposed by Ruff et al. [29]. In contrast to other approaches, this model provides excellent detection performance in experimental settings and can be easily extended to support *semi-supervised* learning, enabling a calibration using synthetic anomalies [29].

Deep SVDD is a two-stage model, where in the first stage an autoencoder (AE) is trained on vector-spaced data. The weights of the AE are then used to initialize the weights $\mathcal{W}$ of a neural network ϕ. Moreover, a center c is set as the mean of the outputs. After training, the anomaly score corresponds to the distance between the embedding and the center. This detection works by training a neural network that minimizes the volume of a hypersphere, which encloses the network's representation of the data [36]. For an input space $\mathcal{X} \subseteq \mathbb{R}^D$ and output space $\mathcal{Z} \subseteq \mathbb{R}^d$, the objective of Deep SVDD can be written as minimizing:

$$\mathcal{L}(\mathcal{W}) = \frac{1}{N}\sum_{i=1}^{N} \|\phi(x_i;\mathcal{W}) - c\|^2 + \frac{\lambda}{2}\sum_{l=1}^{L}\|W^l\|^2, \lambda > 0 \tag{7}$$

Here the function $\phi(\cdot;\mathcal{W}) : \mathcal{X} \to \mathcal{Z}$ denotes a neural network with L layers and weights $\mathcal{W} = \{W^1, \ldots, W^L\}$ and c is the center of the hypersphere. The second term is a regularization. This objective function will try to minimize the distances for normal nodes and thus these will lie close to the center [29].

Semi-supervised one-class learning. The Deep SVDD model can be easily extended to incorporate a few labeled anomalies during optimization, yielding a semi-supervised learner [28]. To this end, the training data of size N is split into B labeled and U unlabeled samples. We re-label the labeled ones as $Y = -1, +1$, where $y_v = +1$ denotes a labeled normal sample while $y_v = -1$ denotes a labeled anomalous sample. Semi-supervised learning enables us to guide the learning process towards characteristics of the graphs relevant for spotting anomalies, such as spatial and temporal relations.

In turn, the objective function of the semi-supervised learner can then be rewritten as follows

$$\begin{aligned}\mathcal{L}(\mathcal{W}) = &\frac{1}{U+B}\sum_{u=1}^{U}\|\phi(x_u;\mathcal{W}) - c\| \\ &+ \frac{\eta}{U+B}\sum_{b=1}^{B}\|\phi(x_b;\mathcal{W}) - c\|^{y_b} + \frac{\lambda}{2}\sum_{l=1}^{L}\|W^l\|^2.\end{aligned} \tag{8}$$

The first term for unlabeled samples is similar to Deep SVDD. However, the loss term for the labeled nodes is weighted by the hyperparameter $\eta > 0$ controlling the balance between both terms. $\eta > 1$ gives more importance to the labeled samples while $\eta < 1$ puts more emphasis on the unlabeled ones. Within the labeled data, a distance of labeled normal instances i.e. $y = +1$ is minimized to the center. For labeled anomalous samples $y = -1$, we penalize the inverse of distances thus mapping those away from the center [28].

Combining with the GVAE. We have seen in Section 3.3 how to embed the nodes of collaboration graphs in a Euclidean latent space. Now in the second stage, the decoder of the GVAE is replaced by a classifier that minimizes the distances from normal embeddings to the center and maximizes the distances between anomalous nodes and the center. After training the GVAE and retrieving the latent embeddings Z, we compute the center c as the mean of the embeddings obtained from the first stage of the GVAE. These are then used to train the hypersphere classifier through backpropagation using the aforementioned objective Function (8), where $\phi(\cdot;\mathcal{W}) : \mathcal{X} \to \mathcal{Z}$ is modeled by the GVAE encoder. The resulting semi-supervised one-class learner is finally trained end-to-end as depicted in Figure 3.

In summary, the idea of our learning model is to minimize the distance to the center for the embeddings of normal nodes and maximize the distance to those of anomalous nodes. The inferred distances to the center can then be used as anomaly scores per node and identify backdoors injected into the repository.

4 DATASETS

To investigate the detection performance of our approach, we introduce a dataset extracted from public Github repositories.This dataset is composed of two curated collections of repositories: The first collection is a training corpus augmented with synthetic anomalies for calibrating our approach, while the second partition is comprised of real-world backdoors found in Git commits and used for measuring the detection performance under realistic conditions.

4.1 Training and Calibration Corpus

To construct a dataset for training and calibrating our method, we follow the experimental setup of Gonzalez et al. [12]. In particular, we retrieve 90 repositories from Github with a majority associated with Node Package Manager (NPM) packages with at least 100

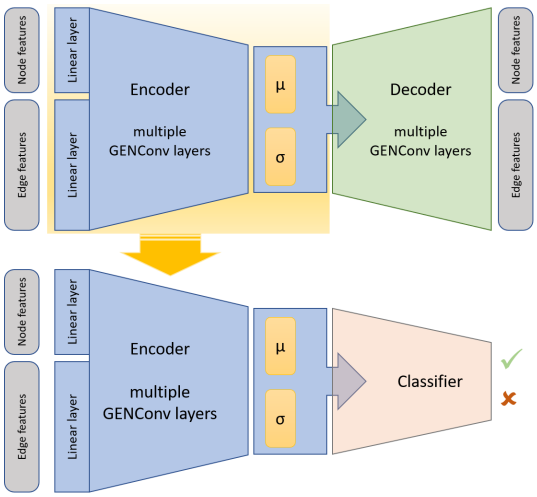

Figure 3: Our neural network architecture for anomaly detection in collaboration graphs.

Table 1: Statistics of training and calibration corpus.

	Developers	Commits	Files	Methods	Anomalies
Mean	99	839	754	1957	555
Stdev	139	1150	2021	3855	664
Max	982	8105	15281	24657	2308
Median	54	389	187	404	301
Min	3	123	10	3	6

commits [12]. The statistics for the collected repositories can be seen in Table 1.

Since our semi-supervised learner needs to be calibrated, it is necessary to apply a handful of labeled data to calibrate the center c. Using the sparsely available real backdoors for this calibration could lead to overfitting and over-optimistic results. Hence, we apply a data augmentation strategy by injecting artificial anomalies into the repositories to enable the model to detect unusual activities in the collaboration graph. Note that these artificial anomalies are not representative of all possible anomalies. Yet they serve as a reference for calibrating our approach. Compared to a fixed manually curated rule-set, this approach significantly reduces bias. We define five types of anomalies that deviate from normal commits through perturbations of the structure and attributes of the collaboration graph similar to the work by Ding et al. [5].

Type 1: Spoofed authorship. The first artificial anomaly type simulates malicious actors who purposefully manipulate the authorship. It is crafted by copying a commit along with its method and file nodes to the head of the repository. Developers are randomly connected to this new commit.

Type 2: Spoofed topology. This anomaly type helps the model to focus on the structural properties of a commit and enables it to maintain a coherent view of a repository. We create a copy of a commit and replace all of its nodes' features with features from randomly picked commit nodes from the same repository. Both the commit to be copied and the one replaced are selected uniformly at random. The edge features of the new commit remain unchanged.

Type 3: Spoofed time features. The third anomaly type simulates malicious actors who purposefully manipulate the commit-date information. We create a copy of a commit and only replace the node features derived from timestamp information with the time features from a randomly picked commit of the same repository. Both the commit to be copied and the one replaced is selected randomly. Edge features of the commit remain unchanged.

Type 4: Spoofed message. This anomaly type will help the model to detect semantic discrepancies between the communicated intention and the actually introduced changes of a commit. We create a copy of a commit and replace only the word2vec embedding derived from the commit message with a randomly picked commit from the same repository. Both commits are selected randomly.

Type 5: Spoofed numerical features. The final anomaly type helps the model to establish a link between numerical and structural features and to learn a relationship between the preceding and subsequent commits. We randomly select a node and replace its features with those of the node that has the largest distance to it. The types of both nodes need to be the same, for instance, a commit can only be swapped with another commit, a file only with another file and so on.

We include these five types of anomalies in the 90 collected Github repositories to create a training and calibration corpus. The number of injected anomalies forms another parameter subject to optimization. A large relative number of anomalies will cause the model to overfit to the synthetic data, while conversely, a too-small number will cause the model to largely ignore the anomalous data. Consequently, we determine the optimal ratio of anomalies from $\{0.1\%, 0.5\%, 0.8\%, 1\%, 1.5\%, 2\%, 3\%\}$ by investigating the detection performance on the training corpus.

4.2 Testing Corpus

To evaluate our approach under realistic conditions, we introduce a testing corpus. We retrieve 19 Github repositories containing nine different types of software backdoors. The statistics of the collected repositories are reported in Table 2.

Table 2: Statistics of testing corpus.

	Developers	Commits	Files	Methods
Mean	18	269	320	632
Stdev	23	637	675	1588
Max	80	2295	2778	6783
Median	3	25	33	72
Min	1	1	8	2

One repository contains the PHP backdoor shown in Listing 1 and five repositories contain a backdoored node-js NPM module. Another two commits refer to the Gentoo and Systemd repositories, that were compromised in 2018[2] leading to several maliciously placed commits. Lastly, we identified several Github repositories

[2]https://wiki.gentoo.org/wiki/Project:Infrastructure/Incident_reports/2018-06-28_Github

infected by the malware *Octopus Scanner*[3]. This malware abuses a compromised host to search for NetBeans projects where it includes a copy of itself. It infects jar files, disguises itself as `cache.dat` and instruments the NetBeans configuration to execute itself whenever a developer builds the project. Interestingly, the malware thereby enters commits of the developed software and hence resembles another variant of a software backdoor.

5 EMPIRICAL EVALUATION

We proceed to empirically evaluate the detection performance of our approach on the collected datasets. We use three baselines to compare our approach: a "shallow" one-class support vector machine (One-class SVM), a statistical approach by Goyal et al. [14] and a rule-based approach by Gonzalez et al. [12] for detecting anomalous commits.

5.1 Experimental Setup

Before presenting the results of our evaluation, we first introduce the experimental setup for our approach and the three baseline methods. In addition, we describe implementation details for the different detection methods that help to reproduce the experiments and expand our methodology.

Setup of our approach. We train the semi-supervised one-class learner described in Section 3 using the training corpus on all 90 repositories with an injection rate of 2% for each anomaly type. We normalize the overall distances and use one standard deviation as the threshold for the commits to be labeled anomalous. Using a hyperparameter optimization on a 70/30 validation split on the training corpus, we choose a learning rate of 0.001 for the GVAE and a learning rate of 0.01 for the one-class learner. Both components are trained using the ADAM optimizer with a training batch size of 128 graphs and an η of 0.75.

One-class SVM. We compared several further baselines on the artificial anomalies with three-fold cross-validation. As depicted in Table 4, the one-class SVM baseline achieves the best results and builds on a popular shallow-learning strategy for anomaly detection. The one-class SVM is trained using all 90 repositories and then tested on the dataset containing the real backdoors. Compared to our model, the SVM does not consider the graph structure, however, it uses the same pre-processed features and thus can also spot some types of anomalies in the data.

Table 4: Comparison of different baselines.

Model	AUC-Score
One-class SVM	**0.70** ±0.04
Deep-SVDD	0.61 ±0.13
Local Outlier Factor	0.53 ±0.00
Elliptic Envelope	0.50 ±0.00
Isolation Forest	0.49 ±0.09

Rule-based detection. As the second baseline, we implement the rule-based approach called *Anomalicious* by Gonzalez et al. [12]. The approach uses commit logs and repository metadata to automatically detect anomalous and potentially malicious commits. This detection is conducted using a comprehensive set of manually engineered detection rules. Unfortunately, the authors do not provide their source code. Hence, we re-implement these rules and publish the reproduced code with this paper for future research.

The method uses thresholds fixed in a specific optimization procedure: Each threshold is selected by flagging a minimum amount of commits but at least one as anomalous. The authors optimize these hyperparameters on a set of 100 repositories. We could only obtain 90 of them, as some of the considered repositories are not available on Github anymore. However, our dataset is similar in size and structure and our testing corpus comprises more recent backdoors.

Statistical detection. As the third and last baseline, we employ the *Unusual Commit Detector* by Goyal et al. [14]. Their approach builds a distribution over past commit features including the commit message lengths, times, changed files and their respective file types. By modeling such distributions they are able to infer ownership of files and detect statistical outliers solely by looking at the commit's past metadata. We stick to their original implementation[4]. Using this approach, a commit property is considered unusual if it deviates from at least 90% of the features of past commits. The feature scores are averaged to obtain a single threshold.

We implement the commit extraction for all models using PyDriller [24] and the GitHub Rest API [11]. Our GVAE and hypersphere classifier model are implemented on top of Pytorch Geometric [25]. We run our experiments on AWS EC2 g4dn instances with CUDA optimization enabled. All experiments are repeated 16 times with different seeds.

Performance Measures. We compare the models using the true-positive rate (TPR) and false-positive rate (FPR) over the number of commits per repository. The true-positive rate indicates the successfully flagged anomalies, while the false-positive rate measures the benign commits that are flagged as anomalous. The base models of Gonzalez et al. [12] and Goyal et al. [14] classify only commits, unlike our model, which can detect anomalies in any type of node. For a better comparison, in our experiments, we thus focus on anomalous commit nodes. The distances of other node types transitively contribute to the commit node's distance since our model uses the message passing aggregation scheme. We further calculate the macro-average as the false positives per repository averaged over all 19 repositories and the micro-average as the false positives weighted by the number of commits per repository. The Receiver Operating Characteristic curve gives a notion about the true positive and false positive rate, hence we calculate the Area under Receiver Operating Characteristic (AUROC) when classifying all node types.

5.2 Detection Performance

To provide an intuition for the difficulty of detecting anomalous commits, we start by first looking at the performance of our approach on the training corpus. Our detection method yields a mean AUROC of 0.859±0.02, a TPR of 0.883±0.06 and an FPR of 0.220±0.17

[3]https://securitylab.github.com/research/octopus-scanner-malware-open-source-supply-chain/

[4]https://github.com/goyalr41/UnusualCommitExtension

Table 3: The detection performance of our approach and the baselines on the testing corpus.

	Repository (https://github.com/)	Commits	Type	Gonzalez et al., 2021		One-class SVM		Goyal et al., 2018		Our Model	
				Found?	FPR (%)	Found?	FPR (%)	Found?	FPR (%)	Found?	FPR (%)
1	atom-minimap/minimap	2,295	CWE-912 Backdoor Injection	✗	3.53	✗	56.50	✓	15.23	✗	0.13
2	greatsuspender/thegreatsuspender	1,536	CWE-512 Backdoor Injection	✗	5.08	✗	8.81	✗	4.18	✗	1.10
3	gentoo/gentoo (25.06.2018 to 29.06.2018)	1,284	CWE-511 Malicious Code	✗	9.28	✓	39.01	✗	6.02	✓	3.23
4	RIAEvangelist/node-ipc	406	CWE-511 Malicious Code	✗	17.28	✗	8.33	✗	7.01	✗	2.04
5	php/php-src (15.03.2021 to 15.04.2021)	390	CWE-510 Backdoor Injection	✓	25.00	✗	1.28	✗	0.00	✓	8.25
6	dominictarr/event-stream	322	CWE-912 Backdoor Injection	✗	13.71	✓	100.00	✗	10.04	✓	0.90
7	Marak/colors.js	259	CWE-511 Malicious Code	✗	6.43	✗	3.12	✗	5.04	✗	4.20
8	gentoo/systemd (25.06.2018 to 29.06.2018)	91	CWE-511 Malicious Code	✓	7.78	✗	4.11	✗	6.02	✓	7.12
9	TesyarRAz/KeseQul-Desktop-Alpha	35	CWE-507 Malware	✗	5.88	✓	100.00	✗	29.41	✓	55.88
10	SebasR16/BdProyecto	15	CWE-507 Malware	✓	0.00	✓	100.00	✗	7.14	✓	71.43
11	Sliray/Secuencia-Numerica	7	CWE-507 Malware	✓	66.67	✓	100.00	✓	16.67	✓	50.00
12	KosimCorp/RatingVoteEPITECH	5	CWE-507 Malware	✗	25.00	✓	100.00	✗	20.00	✓	0.00
13	george-bennett/V2Mp3Player	5	CWE-507 Malware	✗	0.00	✓	100.00	✓	0.00	✓	0.00
14	KosimCorp/Kosim-Framework	1	CWE-507 Malware	✗	0.00	✓	0.00	✗	0.00	✓	0.00
15	SebasR16/Punto-de-venta	1	CWE-507 Malware	✓	0.00	✓	0.00	✗	0.00	✓	0.00
16	BarbosaO/2D-Physics-Simulations	1	CWE-507 Malware	✗	0.00	✓	0.00	✗	0.00	✓	0.00
17	SierraBrandt/GuessTheAnimal	1	CWE-507 Malware	✗	0.00	✓	0.00	✗	0.00	✓	0.00
18	callmehetch/PacmanGame	1	CWE-507 Malware	✗	0.00	✓	0.00	✗	0.00	✓	0.00
19	FelixGtz99/ProyectoGerundio	1	CWE-507 Malware	✗	0.00	✓	0.00	✗	0.00	✓	0.00
Summary		6,656		5/19		13/19		3/19		**15/19**	
Macro-Average					9.77		37.95		**6.68**		10.75
Micro-Average					3.96		24.57		6.97		**0.70**

for the anomaly detection task on the synthetic dataset with injected anomalies. We observe that the model performs very well in detecting Anomaly Type 1 and 5. We assume that these anomaly types are too trivial for the model to distinguish from normal commits. Since Anomaly Type 1 was developed specifically to detect tampering with authorship, we conclude that the model is capable of inferring ownership of files and methods. For example, whether a contributor was the one who created a file or method and whether she was the only one who touched it in the past. Type 5 randomly swaps two maximally distant commits, indicating that the model is able to associate node features with their topological properties. Type 4 is the worst-performing anomaly type, where the model detects the fewest anomalies with only 65% accuracy. This shows that linking natural language with surrounding features is difficult. We conclude that the synthetic anomalies are sufficiently diverse for calibrating our model and thus proceed to inspect its performance on the testing data.

Comparison with Gonzalez et al. [12]. In Table 3 we present the 19 real backdoors found in public Git repositories. The projects are sorted by their number of commits. The PHP, *Gentoo* and *systemd* samples were taken over a smaller time frame since the repositories are fairly large and each commit potentially touches hundreds of files. In the table, we can see that our model detects 15 out of 19 backdoors, compared to the baseline from Gonzalez et al. [12], which only detects 5 out of 19. Overall, our model shows a lower FPR averaged over the 6656 commits, however, a higher FPR averaged over the 19 projects.

Both models detect a maliciously placed piece of code in the *systemd* repository[5]. The commit says that the configure script has been fixed, but when run by an innocent developer, her home directory is deleted. The commit is highly suspicious because the committer has never committed to the repository before and, on top of that, he manipulated a rarely touched file. The baseline by Gonzalez et al. [12] fails to detect several Octopus Scanner repositories, while they are all detected by our model. The projects with the injected backdoors are harder to detect, however, both models found the recent PHP backdoor at the cost of a high (25%) FPR for the baseline whatsoever. Our model also detects a malicious commit in *event-Stream*[6] with a low (0.9%) FPR. In this commit, an unnecessary and malicious dependency was introduced capable to steal the developer's Bitcoin wallet's private keys. Just as for Gonzalez et al. [12], we can observe a lower FPR with an increasing number of commits per project. This makes sense since larger projects offer more information to the inference process.

Comparison with Goyal et al. [14]. We further compare our model against the statistical approach by Goyal et al. [14]. Their model only reports 3/19 anomalies correctly. Using their approach with deviating nominal features, they achieve the lowest macro-average FPR. Unfortunately, their tool is not able to reason about initial and merge commits, making it unsuitable for the comparison of the single-commit-repositories infected with the Octopus Scanner. Our model is able to check initial commits since we pre-train it on a large training corpus. This way, we are able to reason about one repository using the knowledge inferred from others.

Both models detect the Octopus Scanner in the *V2Mp3Player* and *Secuencia-Numerica* repositories. Goyal et al. [14] even reports a lower FPR for the latter. Considering repository *Secuencia-Numerica*, their tool explains the outlier because more LoC than 90% of the other commits were changed, only 0.3% other commits added more LoC and more files were touched than 90% of the other commits. The most important identification however is, that `.dat` files were changed and such files are rarely changed in the repository with less than 2% of all touched file types. The last explanation involves the Octopus Scanner itself, however, only in combination with the other unrelated anomalous properties, the commit could be flagged. Furthermore, in 2017, advertising popups were injected into the open-source software *minimap*. Goyal et al. [14], in contrast

[5]https://github.com/gentoo/systemd/commit/50e3544d

[6]https://github.com/dominictarr/event-stream/commit/e316336

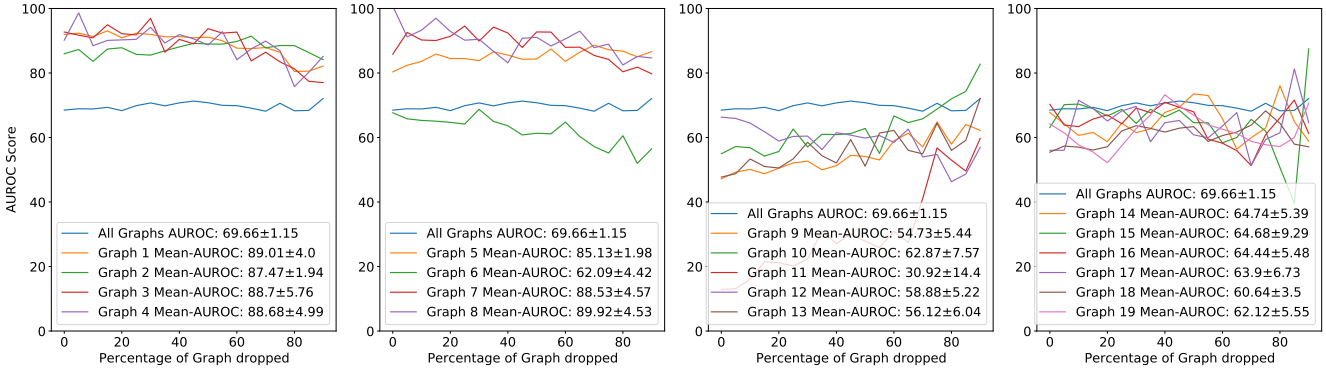

Figure 4: Robustness is measured as the drop in the AUROC score for increasing graph perturbations.

to our model, detects this. The explanation is that this particular commit touches `.js` and `.less` files in conjunction which happens rarely in this repository. The `.less` file is associated with the popup banner's style sheet for the injected advertisement. The baseline fails to detect other malicious commits, however, it pertains to an overall low FPR.

Comparison with one-class SVM. In Table 3 we can also see that the one-class SVM beats the rule-based and statistical model measured by the detection rate. However, the false-positive rate is nearly 400% as high as compared to the rule-based model. For instance, relying on the one-class SVM, a security practitioner would need to review more than half of the *minimap* repository and still would not find the actual malicious commit. Furthermore, the model collapses for graphs with fewer commits, since from repository 9 to 19 everything is labeled as anomalous. The FPR decreases starting with the *systemd* repository at around 90 commits. Since the baseline builds upon the same feature set as our method, we conclude that the graph structure is fundamentally important for spotting malicious commits in the version history of software projects.

5.3 Robustness against Adversaries

Additionally, we measure the robustness of our approach. On the one hand, we want to show to what extent the graph topological structure influences anomaly detection and on the other, we want to assure that an adversary with knowledge of our model needs to put in some effort to evade it.

Measuring the resiliency against potential adversaries is hard and to avoid introducing any bias, we stick to an approach by Warnecke et al. [40]. We evaluate the model's robustness by removing random clean and malicious nodes from the collaboration graphs to observe whether the model is still able to detect the anomalies.

We measure the AUROC to examine how well the model performs when commits are perturbed until they deviate from the learned distribution in Figure 4. Further, we observe how much a malicious commit has to be perturbed until the model fails to detect it. Practically speaking, a potential attacker would first craft a malicious commit and then, to evade detection, decrease the difference in appearance to the past clean commits in the repository. We simulate this by dropping nodes until the difference between the structural properties of the clean commits and the deviating commit vanishes and therefore is harder to distinguish.

A downward trend indicates that noise and manipulations in the graph structure affect the model more. Conversely, a constant or upward trend hints that the model is robust against graph perturbations. Since most nodes in a collaboration graph are benign, their removal results in a lower number of false positives, leading to an upward trend. A drop rate of 100% removes nodes from all types with the only exception of the commit nodes. We also report the mean AUROC for each graph averaged over all perturbations for 16 runs. Each graph is numbered and corresponds to the respective project from Table 3.

The first two plots from the left show the model's performance for the collaboration graphs of the first eight larger repositories, presenting a downward trend with increasing drop probability. Apart from the noise, the curves from the smaller collaboration graphs on the two right plots show a slight upward trend. Graph 11 is an outlier since it presents an upward trend starting from an initial low AUROC. Since this graph already induces as much as 50% FPR and removing nodes consequently also removes FPs. The legend shows that the larger repositories have an overall higher AUROC score than the repositories with fewer commits. We further observe that the model is robust against graph manipulations for the smaller projects and slightly less robust for the larger projects containing real malicious commits. This could be due to the fact, that with only a few commits (< 35) present in the repository, the model prioritizes the node features over the graph structure. Moreover, the graph structure has a greater influence on the model's decision for repositories with more (> 100) commits.

5.4 Interpretability

Figure 5 depicts a repository[7] augmented with synthetic anomalies. Each dot represents a node in the collaboration graph, while the plot is divided up into four sections. On the left, the nodes belong to the developer, followed by the commits and finally, the darker shaded sections represent the nodes belonging to the category of files and methods. Blue dots represent nodes found in the original repository, while different colored nodes represent different anomaly types.

[7] https://github.com/aspnet/SignalR

For visualization purposes, we remove the anomaly types that are too trivial to detect, that is, Type 1 and Type 5.

The figure shows each node with its associated distance to the center. Most injected anomalies are further away from the center, while the majority of normal nodes are placed within a distance of around 0.7 to the hypersphere center. This can be attributed to the fact that certain anomalies are hardly distinguishable from the normal nodes and by Equation (8) they tend to pull their entire neighborhood away from c. A security practitioner can easily create such plots and examine the nodes that have been placed outside the border of the hypersphere. For example, in Figure 6, we present an exemplary false-positive commit extracted for further investigation from Figure 5. The commit refers to a merge commit[8]. For visualization purposes, we substitute the file, method and developer names. The graph structure reveals that the anomalous (red) commit node has two parents and thus represents a merge commit. Although the graph does not expose any malicious intent from the developer, it, however, shows that changes only from one of its two parent commits were merged, ignoring the files and methods from the other. This does not pose any anomaly in a security-critical sense, however, it is an unusual development behavior and should be examined anyhow.

Moving on with the analysis of our testing data, we can easily analyze the samples in the same way for, e.g., detecting the PHP backdoor. The model identifies the subsequent commit[9], which also constitutes a backdoor, with one standard deviation distance to the center c. The commit is unsigned, thus, the impersonated author *Nikita Popov* of the second backdoor has attributed a distance of three standard deviations to c. The file `zlib.c` containing the backdoors from Figure 1 and the subsequent backdoor, has a standard deviation of two. Overall the model flags both commits

[8]https://github.com/aspnet/SignalR/commit/7de91da
[9]https://github.com/php/php-src/commit/2b0f239

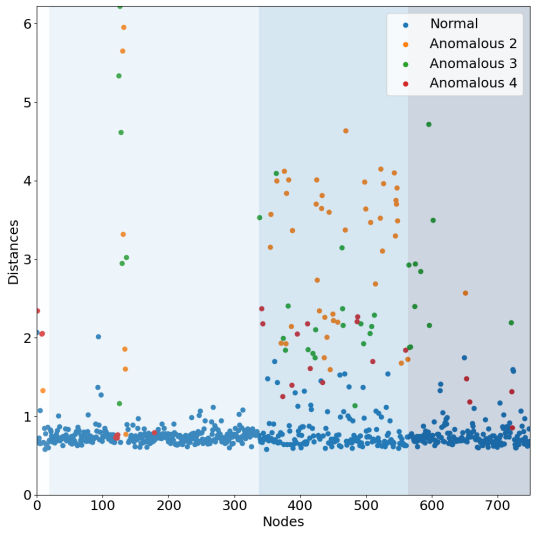

Figure 5: Visualizing anomalous nodes in a collaboration graph (developers, commits, files, methods).

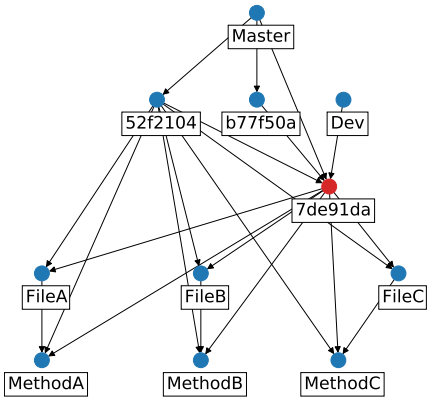

Figure 6: Collaboration graph labeled as false positive.

as anomalies. The Octopus Scanner repositories are detected according to their `cache.dat` similar to the approach by Goyal et al. [14]. The backdoor in *event-stream*, for instance, is found due to the modification in the `package.json` with a distance of three standard deviations to c. The backdoors in *Minimap* and *Thegreatsuspender* were intentionally planted by the maintainer, leading our model to have low confidence for suspecting the author node.

5.5 Practicability

Although code reviews seem to be a solution against supply chain attacks, Rigby et al. state that many contributions to open source software (OSS) do not receive any review at all. Attention from the community is a sparse resource and not every commit might receive the attention it needs [27]. Since reviewing each commit thoroughly might be too costly and never reviewing a commit too risky, we suggest to deploy our model to pre-filter commits that might need stronger attention. Inspired by the work from Saidani et al. [31] we can impose a cost model to help us decide when our detection model is practicable.

We use the conventional terminology for describing the quality of the models using true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN). Furthermore, we assume that there is a cost r for reviewing a single commit and a cost d if a malicious commit sneaks through. Clearly, the cost the review is much lower than the cost for an undetected backdoor, hence $r \ll d$. Still, both r and d might be calculated as some form of financial cost, for example, person-hours.

Cost models. We proceed to define three theoretical cost models to analyze the utility of our approach and the baselines methods. We start with the cost model for a detection method:

$$cost_{\text{DETECT}} = r \cdot (FP + TP) + d \cdot FN$$

Alternatively, one might also review every commit in a software repository. In this case, no detection method is needed and we arrive at the following cost model:

$$cost_{\text{ALWAYS}} = r \cdot (FP + TP + FN + TN)$$

Table 5: Cost comparison $\frac{r}{d}$ (in %) for the models against two baseline strategies.

Model	Beats Always	Beats Never
Our Model	$>$ **0.06**	$<$ **24.21**
Gonzalez et al. [12]	$>$ 0.22	$<$ 1.87
Goyal et al. [14]	$>$ 0.10	$<$ 0.64

Finally, we can ignore the risk of software backdoors and simply never review any commit in a repository. In this case, we obtain the following cost model:

$$cost_{\text{Never}} = d \cdot (FN + TP)$$

By putting these cost models in relation, we can construct boundary conditions that help us assess the utility of detection methods. First, it follows that we have $cost_{\text{Detect}} < cost_{\text{Always}}$ if:

$$\frac{r}{d} > \frac{FN}{FN + TN} \quad \text{false omission rate (FOR)}$$

Similarly, we can construct a boundary for not reviewing, and we get $cost_{\text{Detect}} < cost_{\text{Never}}$ if:

$$\frac{r}{d} < \frac{TP}{FP + TP} \quad \text{positive predictive value (PPV)}$$

Note that both the false omission rate and the positive predictive value are monotonically increasing in terms of the fraction of real positive cases (actual flaws) in the data, i.e. the prevalence.

Cost analysis. Taking the empirical results for each detection method from Table 3 we get the following values for the cost models: $FP = 47$, $TP = 15$, $FN = 4$ and $TN = 6590$. Since we have two boundary conditions we can explicitly calculate the threshold $\frac{r}{d}$ for all models against both baseline strategies. As depicted in Table 5, our approach outperforms $cost_{\text{Always}}$ if $\frac{r}{d} < 0.06\%$ and we beat $cost_{\text{Never}}$ if $\frac{r}{d} < 24.21\%$. Moreover, our model has a large favorable range compared to the baseline models by Gonzalez et al. and Goyal et al. [12, 14], making it practicable in more situations. Only if the cost of an review compared to that of an undetected backdoor is really high Never review any code is a promising strategy. On the other hand, $\frac{r}{d}$ must be very low in order to make reviewing Always a winning strategy over our learning-based approach.

To put this analysis in context, let us consider this hypothetical example: Suppose that the cost of reviewing a commit in a software project is 2 person-hours. According to recent statistics from the UK, an average data breach costs $\sim$ 1400 USD[10] while the average developer might cost 25 USD per hour[11]. Hence, we have $\frac{r}{d} = 3.57\%$ which is well inside the boundaries calculated above and hence our approach is better than both baseline strategies. Moreover, since 3.57% exceeds both 1.87% and 0.64% the other models are more expensive than the Never strategy and therefore not practicable. Concluding, our model is the preferable choice compared against the two other baseline models by Gonzalez et al. and Goyal et al. [12, 14] under practical conditions.

[10]https://www.statista.com/statistics/586788/average-cost-of-cyber-security-breaches-for-united-kingdom-uk-businesses/
[11]https://uk.talent.com/salary?job=software+engineer

6 RELATED WORK

Anomalies can be defined as observations in the data that do not conform to expected behavior. Anomaly Detection refers to the task of finding such observations [3]. Several works have been conducted to detect anomalies in graph-based datasets, for instance, the work by Ding et al. [5], Liu et al. [22] or Dou et al. [7], all leverage graph neural networks with promising results.

Some significant work has already been done concerning vulnerability detection on code graphs. A GNN-based model beating popular non-learning and non-graph-learning-based static analyzers is used by Zhou et al. [44] for graph-level classification using manually labeled real-world datasets. It has been demonstrated by Chakraborty et al. [2] that the effectiveness of the state-of-the-art deep learning-based vulnerability detection models significantly deteriorates when applied to unseen real-world datasets. Some pitfalls in learning-based vulnerability detection models are identified by Arp et al. [1], where they also suggest possible remedies.

Collaboration graphs have been already subject to analysis in conjunction with graph-based machine learning. Collaboration graphs were introduced by GraphRepo [15] to enable developers to efficiently query through commits using graph databases. Geiger et al. [9] mine commit graphs with additional metadata from android project repositories and further introduce a large collaboration graph dataset. Dong [6] use graph neural networks to learn commit representations to suggest meaningful commit messages based on the proposed code changes. Similarly, Shen et al. [34] decompose commit graphs based on the coherence of the underlying changes. None of these approaches, however, focuses on the detection of backdoors in the graphs.

To counteract the impersonation of authors and to easier detect maliciously crafted commits, Git allows signing off commits by their respective authors[12]. Torres-Arias et al. introduce a security scheme on top of Git to evade metadata tampering in Git repositories when several developers work on conflicting views of the repository [38]. Li et al. and Feldman et al. enable developers to sign off the state of a repository enforcing fork-consistency on potentially untrusted servers [8, 21]. All these techniques provide overhead to either the infrastructure or the collaboration workflow. We offer an orthogonal approach to detect anomalous commits without interfering with Git.

7 CONCLUSION

In this work, we show that current methods for detecting anomalies in Git commits are inadequate. These models are driven by manually created rules that limit detection capabilities, generalization and introduce significant bias into empirical evaluations in their respective works.

In contrast, we port a popular architecture for unsupervised anomaly detection to the graph-learning domain. Using recent advances in graph neural networks and unsupervised outlier detection, we can leverage collaboration graphs to detect anomalous commits. By modeling project repositories as collaboration graphs, we can analyze new properties that arise from their topological structure. For example, such graphs reveal common collaborations between developers, files and methods that are frequently modified

[12]https://git-scm.com/book/en/v2/Git-Tools-Signing-Your-Work

together, or commits that influence other commits. Instead of manually curated rules, we rely on a loose set of artificial anomalies to calibrate an unsupervised model.

Our model outperforms current state-of-the-art models while being able to detect not only anomalous commits, but also files, methods, and suspicious developers. In this work, we show that our model is robust, interpretable, practicable and capable of detecting notable past malicious supply chain attacks, for example, the backdoor introduced to the PHP repository in 2021 or recently compromised NodeJS packages.

ACKNOWLEDGMENTS

This work has been funded by the Federal Ministry of Education and Research (BMBF, Germany) in the project IVAN (FKZ: 16KIS1165K).

REFERENCES

[1] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2022. Dos and Don'ts of Machine Learning in Computer Security. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity22/presentation/arp

[2] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2020. Deep Learning based Vulnerability Detection: Are We There Yet? arXiv:2009.07235 [cs.SE]

[3] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly Detection: A Survey. *ACM Comput. Surv.* 41, 3, Article 15 (jul 2009), 58 pages. https://doi.org/10.1145/1541880.1541882

[4] Marco di Biase, Ayushi Rastogi, Magiel Bruntink, and Arie van Deursen. 2019. The Delta Maintainability Model: Measuring Maintainability of Fine-Grained Code Changes. In *Proceedings of the Second International Conference on Technical Debt* (Montreal, Quebec, Canada) *(TechDebt '19)*. IEEE Press, 113–122. https://doi.org/10.1109/TechDebt.2019.00030

[5] Kaize Ding, Jundong Li, Rohit Bhanushali, and Huan Liu. 2019. *Deep Anomaly Detection on Attributed Networks*. 594–602. https://doi.org/10.1137/1.9781611975673.67

[6] Jinhao Dong. 2022. Reproducible Package of FIRA: Fine-Grained Graph-Based Code Change Representation for Automated Commit Message Generation. Zenodo. https://doi.org/10.5281/zenodo.6053202

[7] Yingtong Dou, Zhiwei Liu, Li Sun, Yutong Deng, Hao Peng, and Philip S. Yu. 2020. Enhancing Graph Neural Network-Based Fraud Detectors against Camouflaged Fraudsters. In *Proceedings of the 29th ACM International Conference on Information 'I&' Knowledge Management* (Virtual Event, Ireland) *(CIKM '20)*. Association for Computing Machinery, New York, NY, USA, 315 to 324. https://doi.org/10.1145/3340531.3411903

[8] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. 2010. SPORC: Group Collaboration using Untrusted Cloud Resources. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/osdi10/sporc-group-collaboration-using-untrusted-cloud-resources

[9] Franz-Xaver Geiger, Ivano Malavolta, Luca Pascarella, Fabio Palomba, Dario Di Nucci, and Alberto Bacchelli. 2018. A Graph-Based Dataset of Commit History of Real-World Android apps. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 30–33.

[10] Gensim. 2022. Gensim Word2Vec. https://radimrehurek.com/gensim/models/word2vec.html

[11] GitHub. 2022. GitHub Rest API. https://docs.github.com/en/rest

[12] Danielle Gonzalez, T. Zimmermann, Patrice Godefroid, and Maxine Schaefer. 2021. Anomalicious: Automated Detection of Anomalous and Potentially Malicious Commits on GitHub. *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (2021), 258–267.

[13] Dan Goodin. 2021. Hackers backdoor PHP source code after breaching internal git server. News Posts, ArsTechnica.

[14] Raman Goyal, Gabriel Ferreira, Christian Kästner, and James Herbsleb. 2017. Identifying unusual commits on GitHub: Goyal et al . *Journal of Software: Evolution and Process* 30 (09 2017), e1893. https://doi.org/10.1002/smr.1893

[15] GraphRepo. 2021. GraphRepo. https://graphrepo.readthedocs.io/en/latest/index.html

[16] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 1025–1035.

[17] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *arXiv preprint arXiv:2005.00687* (2020).

[18] Diederik P. Kingma and Max Welling. 2019. An Introduction to Variational Autoencoders. *CoRR* abs/1906.02691 (2019).

[19] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations (ICLR)*.

[20] Guohao Li, Chenxin Xiong, Ali Thabet, and Bernard Ghanem. 2020. DeeperGCN: All You Need to Train Deeper GCNs. arXiv:2006.07739 [cs.LG]

[21] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. 2004. Secure Untrusted Data Repository (SUNDR). In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/osdi-04/secure-untrusted-data-repository-sundr

[22] Zhiwei Liu, Yingtong Dou, Philip S Yu, Yutong Deng, and Hao Peng. 2020. Alleviating the inconsistency problem of applying graph neural network to fraud detection. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 1569–1572.

[23] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer International Publishing, Cham, 23–43.

[24] PyDriller. 2021. PyDriller. https://pydriller.readthedocs.io/en/latest

[25] PyTorch Geometric. 2022. PyTorch Geometric. https://pytorch-geometric.readthedocs.io/en/latest/

[26] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74 (1953), 358–366.

[27] Peter C. Rigby, Daniel M. German, Laura Cowen, and Margaret-Anne Storey. 2014. Peer Review on Open-Source Software Projects: Parameters, Statistical Models, and Theory. *ACM Trans. Softw. Eng. Methodol.* 23, 4, Article 35 (sep 2014), 33 pages. https://doi.org/10.1145/2594458

[28] Lukas Ruff, Robert A. Vandermeulen, Nico Görnitz, Alexander Binder, Emmanuel Müller, Klaus-Robert Müller, and Marius Kloft. 2020. Deep Semi-Supervised Anomaly Detection. In *International Conference on Learning Representations*. https://openreview.net/forum?id=HkgH0TEYwH

[29] Lukas Ruff, Robert A. Vandermeulen, Nico Görnitz, Lucas Deecke, Shoaib A. Siddiqui, Alexander Binder, Emmanuel Müller, and Marius Kloft. 2018. Deep One-Class Classification. In *Proceedings of the 35th International Conference on Machine Learning*, Vol. 80. 4393–4402.

[30] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 757–762. https://doi.org/10.1109/ICMLA.2018.00120

[31] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. 2020. Predicting Continuous Integration Build Failures Using Evolutionary Search. *Information and Software Technology* 128 (08 2020), 106392. https://doi.org/10.1016/j.infsof.2020.106392

[32] Felix Schuster and Thorsten Holz. 2013. Towards reducing the attack surface of software backdoors.. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 851–862. https://doi.org/10.1145/2508859.2516716

[33] Alex Serban, Magiel Bruntink, and Joost Visser. 2020. GraphRepo: Fast Exploration in Software Repository Mining. *CoRR* abs/2008.04884 (2020). arXiv:2008.04884 https://arxiv.org/abs/2008.04884

[34] Bo Shen, Wei Zhang, Christian Kästner, Haiyan Zhao, Zhao Wei, Guangtai Liang, and Zhi Jin. 2021. SmartCommit: A Graph-Based Interactive Assistant for Activity-Oriented Commits. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 379–390. https://doi.org/10.1145/3468264.3468551

[35] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice: Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware.. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*. https://doi.org/ndss2015/firmalice-automatic-detection-authentication-bypass-vulnerabilities-binary-firmware

[36] David MJ Tax and Robert PW Duin. 2004. Support vector data description. *Machine learning* 54, 1 (2004), 45–66.

[37] Sam L. Thomas and Aurélien Francillon. 2018. Backdoors: Definition, Deniability and Detection.. In *Proc. of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 92–113. https://doi.org/10.1007/978-3-030-00470-5_5

[38] Santiago Torres-Arias, Anil Kumar Ammula, Reza Curtmola, and Justin Cappos. 2016. On Omitting Commits and Committing Omissions: Preventing Git Metadata Tampering That (Re)introduces Software Vulnerabilities. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 379–395. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/torres-arias

[39] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. *International Conference on Learning Representations* (2018). https://openreview.net/forum?id=rJXMpikCZ accepted as poster.

[40] Alexander Warnecke, Daniel Arp, Christian Wressnegger, and Konrad Rieck. 2020. Evaluating Explanation Methods for Deep Learning in Security. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, Genoa, Italy, 158–174. https://doi.org/10.1109/EuroSP48549.2020.00018

[41] Qiushi Wu and Kangjie Lu. 2021. On the Feasibility of Stealthily Introducing Vulnerabilities in Open-Source Software via Hypocrite Commits. Technical Report, University of Minnesota.

[42] Chris Wysopal and Chris Eng. 2007. Static Detection of Application Backdoors. *Datenschutz und Datensicherheit - DuD* 34 (01 2007). https://doi.org/10.1007/s11623-010-0024-4

[43] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826* (2018).

[44] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems*, Vol. 32.