PuRe Defender: A Game-Theoretic Pull Request Assignment with Deep RL

Javad Mokhtari Koushyar[1][0009−0004−5998−462X], Mina Guirguis[1][0000−0003−2629−6172], and George Atia[2][0000−0001−7958−9855]

[1] Texas State University, San Marcos TX 78666, USA
{koushyar,msg}@txstate.edu
[2] University of Central Florida, Orlando FL 32816, USA
george.atia@ucf.edu

Abstract. The pull-based model in Open-Source Software (OSS) has enabled decentralized collaboration and major advancements, but it also opens the door to supply-chain attacks, where an attacker submits a malicious pull request with the intention of injecting malicious code into the codebase. Once a malicious pull request is merged, all downstream systems depending on the package may be compromised. This paper investigates the problem of assigning pull requests to maintainers in OSS packages from a game-theoretic standpoint. We model the problem as a two-player (defender and attacker) general-sum game with partial observability in which the attacker submits malicious pull requests while the defender assigns pull requests to available maintainers. The model captures critical features such as the availability of the maintainers, their expertise, and the quantity and severity of the pull requests. Many of those features can be publicly inferred. Accordingly, we develop deep reinforcement learning-based algorithms within the Policy-Space Response Oracle (PSRO) framework to derive potent strategies for both players, addressing the complexity of the formulation and the explosion of state and action spaces. We assess the behavior of the derived policies on two real-world Python packages with different sizes. We show that the policies obtained outperform other assignment strategies.

Keywords: Open Source Software · Software Supply Chain Attacks · Pull Request Assignment · Multi-Agent Reinforcement Learning.

1 Introduction

Motivation: Innovation and rapid technological advances (especially in AI) can be largely attributed to the OSS ecosystems. The ability to access the latest repositories, utilizing them as building blocks, improving them and/or spinning them into novel products, and publishing them have been instrumental in fueling innovation, cutting costs, and achieving a shorter time-to-market. The economic savings for scientific OSS in some areas exceeds 87% compared to equivalent or lesser proprietary software [18]. This has led to a wide adoption of OSS in public and private companies, government and academia. In a recent report, 95% of

entities reported maintaining or increasing their OSS usage, with 33% reporting significant increase [4]. Furthermore, entire business models are established and sustained by supporting OSS (e.g., Red Hat).

The security nature of OSS, however, is a double-edged sword. On one hand, it provides an important transparency feature allowing for public inspection, tracking and auditing of the source code. A recent survey showed that 68% of the respondents believe that OSS is more secure than closed-source software [12]. On the other hand, however, it opens the door to an adversary to inject malicious code into an OSS package. Managing security vulnerabilities in OSS has been and remains a key challenge.

In OSS, entities that are interested in contributing code to the codebase submit Pull Requests (PRs). A PR is simply a proposal to make some change in the package that varies from simply adding a comment to resolving a known issue to adding new features. Every OSS package has a list of maintainers who review PRs and decide on whether to accept the PR and consequently merge the requested change into the codebase or deny it. Some novel techniques have been used by attackers to submit PRs that aim to look benign but under the hood they introduce a component of a larger attack vector (e.g., through hypocrite commits [24]).

The impact of malicious code injections in OSS can be quite devastating based on the popularity of the package and the nature of the exploit. For example, in November 2021 a zero-day vulnerability (known as Log4Shell) was reported in Log4j, a Java-based logging utility that is used by many prominent services such as Amazon Web Services, Cloudflare, iCloud, Minecraft and Twitter [17]. The vulnerability was introduced as a feature enhancement in the package that maintainers failed to detect which allows attackers to run any code on affected systems. This incident has triggered the highest Common Vulnerability Scoring System (CVSS) score of 10 and is still being actively exploited at the time of writing [19]. This incident highlights the severity of such a supply-chain attack with a global impact.

Another critical aspect in OSS is that important publicly available information can be leveraged by the adversary when mounting an attack. For example, the number of maintainers (and their level of expertise), their activity, the typical rate of PR arrivals, and the backlog of PRs can help the attacker *time* the submission of malicious PRs to exploit such information.

Scope of work: In this paper, we model the assignment of PRs to maintainers through a novel game-theoretic framework. We cast the problem as an extensive-form, general-sum game with imperfect information, in which the defender does not observe the attacker's actions directly (i.e., the defender receives batches of PRs but cannot distinguish malicious ones in advance). Our focus is on modeling a stealthy attacker who avoids raising suspicion, rather than one who floods the system with PRs to overwhelm maintainers. Finally, instead of seeking theoretically optimal strategies, we aim to learn practical, reinforcement-learning-based policies that are applicable to real-world open-source projects.

Contributions: In this paper, we make the following contributions:

1. We develop a game-theoretic framework for modeling the assignment of PRs to maintainers in the presence of an attacker who aims to inject malicious code into the codebase. Our formulation captures critical factors such as maintainer availability and expertise, the severity of incoming PRs, and the evolving backlog of PRs.
2. To address the combinatorial complexity of the state and action spaces, we design reinforcement learning algorithms within the Policy-Space Response Oracle (PSRO) framework [11], enabling the computation of strong approximate strategies for both the attacker and the defender.
3. We evaluate the learned strategies against baseline policies using two real-world open-source repositories: Pandapower [23] and NumPy [8].

Paper organization: In Section 2, we put this work in context with other related work. In Section 3, we present our game-theoretic formulation and in Section 4 we present our solution framework. In Section 5, we present our performance evaluation and we conclude with a summary in Section 6.

2 Related Work

2.1 Background

Distributed Version Control Systems (DVCS) such as Git [6] have allowed for the emergence of new software development paradigms. The most popular one is the pull-based model [7] which is vastly employed by social coding platforms like GitHub and bitBucket. Figure 1 shows the workflow in a pull-based model. The process starts by a contributor cloning a repository on his/her local computer. A contributor is a user in the OSS community who proposes changes to enhance the package (e.g., fixing a bug or adding a new feature). The changes are performed locally and then submitted through a PR to be reviewed by maintainers. Maintainers are the ones responsible of maintaining the package and typically include original and core developers. After the PR is received, it gets added to a PR backlog pending a review. The review can be done by the maintainers, in a review session and/or open discussions with the OSS users community. A maintainer with merge-access takes action on the PR by either merging it into the codebase, closing it without merging, or keeping it pending in the backlog. If the PR is merged, the changes made will be applied to the repository through DVCS, otherwise the changes are dismissed.

This work is related to two areas of research: (1) the assignment of PRs to maintainers and (2) supply-chain attacks in OSS. Traditionally, these two areas are studied in isolation of each other. In this work, we merge those two areas by deriving assignment policies that consider the presence of stealthy attackers sending malicious PRs while taking into account the availability and expertise of the maintainers.

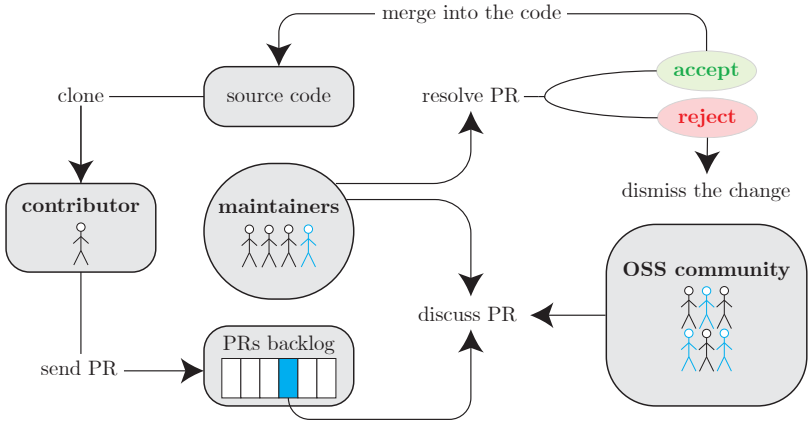

Fig. 1: Pull-based development model workflow

2.2 PR Assignment Problem

There have been a few approaches to the problem of PR assignment and maintainer/reviewer recommendation in OSS that take into account the nature of the PR and/or the maintainers. In the work of de Lima Júnior et al. [13], the authors used a PR classification-based approach to suggest the best maintainer for the PR. In [25], a deep learning model based on CNN and LSTM has been used to learn the features of a PR and then predict the top-k maintainers for that PR. Some other works focused on more specific types of PR assignment. For example, Rong et al. [20] have proposed a recommender system based on a hypergraph over the maintainers which is effective for the cases when multiple maintainers are dealing with a single PR. Other works such as [3] have focused on reducing the workload of the maintainers by computing the probability of merging a PR into the repository and then assigning higher probable PRs to the available maintainers. The above works consider legitimate PRs that may contain random bugs and do not explicitly consider a strategic attacker who is submitting intentional malicious PRs into the workflow.

2.3 Open-Source Software Supply-Chain Attacks

Software supply-chains are vulnerable to a wide range of attacks – from distributing a malicious package through package managers like NPM [1] or PyPI [2], to Typosquatting the name of the packages, to misconfiguring security features to gain merge access of an existing repository. In [16], the authors provide a survey for different supply-chain attacks. Ladisa et al. provided a taxonomy of attacks on OSS supply chains [10]. Within this taxonomy, our work falls under the category of “Inject into Sources of Legitimate Package" and closely relates to the subcategory of “Introduce Malicious Code through Hypocrite Merge Request". There have been some known techniques that attackers can utilize to

inject malicious code into repository. Some of the noticeable examples are exploiting the immature vulnerabilities in the code [24], taking over the accounts of the maintainers [26], and exploiting rendering weaknesses [5], among others.

Other studies have focused on characterizing the process of handling PRs. For example, Khatoonabadi et al. have studied the response time of maintainers in various OSS projects and how it is possible to predict it [9]. Such information can be exploited by an attacker in timing the submission of malicious PRs. Our work combines the above two areas of research through a novel game-theoretic approach to study the adversarial aspects of the PR assignment problem in the presence of an active and stealthy attacker who aims to exploit the knowledge inferred by submitting malicious PRs.

3 Problem Formulation

3.1 Model Overview

We model the PR assignment problem in OSS projects as a two-player extensive-form game between a Defender $\mathcal{D}$ and an Attacker $\mathcal{A}$. The game unfolds over a finite time horizon $T \in \mathbb{Z}^+$, proceeding in discrete time steps. At each time step, players act sequentially: the defender first selects an action (e.g., PR assignments), followed by the attacker, who chooses a batch of malicious PRs to submit. Each state of the game corresponds to a node in an extensive-form game tree (Figure 2). A single time step represents one full cycle in which both $\mathcal{D}$ and $\mathcal{A}$ take their respective actions.

Players: $\{\mathcal{D}, \mathcal{A}\}$, Horizon: $T \in \mathbb{Z}^+$

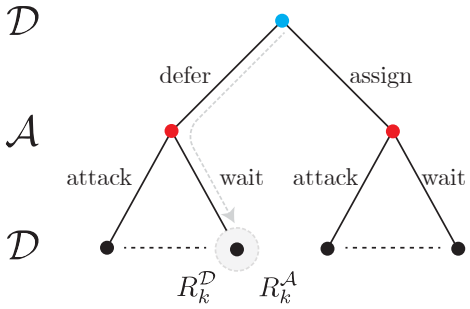

Fig. 2: One time step of malicious pull request assignment game

3.2 Environment and State Representation

The state of each node in the game is defined by the availability of maintainers and the backlog of unresolved PRs in the OSS project. We assume there is an

evolving set of unresolved issues $\mathcal{I}$. The set of maintainers is denoted by $\mathcal{M} = \{m_1, \ldots, m_M\}$. Each maintainer is classified into one of three expertise levels: *Senior*, *Intermediate*, or *Junior*, corresponding to numerical levels 1 through L, where $L = 3$.

Each PR is a structured tuple $\sigma = (i, p, t)$, where $i \in \mathcal{I}$ identifies the issue the PR addresses, $p \in \mathcal{P}_{\text{level}} = \{\text{Urgent}, \text{Normal}, \text{Low}\}$ is the priority level, and $t \in \mathcal{T} = \{\text{benign}, \text{malicious}\}$ is the type of the PR. Benign PRs aim to resolve genuine issues in the source code without introducing additional risk, while malicious PRs are submitted by the attacker in an attempt to inject harmful behavior or backdoors.

Each maintainer has a load value indicating how many time steps remain until they are available. The vector of maintainer loads at time k is denoted by

$$\boldsymbol{\ell}_k = [\ell_1^k, \ldots, \ell_M^k] \in \mathbb{Z}_{\geq 0}^M,$$

where ℓ_j^k is the number of steps until maintainer m_j becomes available. A maintainer is considered available at time k if $\ell_j^k = 0$.

The backlog of unresolved PRs at time k is denoted by $\mathcal{B}_k \subseteq \mathcal{P}$, where $\mathcal{P} = \mathcal{I} \times \mathcal{P}_{\text{level}} \times \mathcal{T}$ is the space of all possible PRs. The full state of the system at time step k is defined as

$$s_k = (\boldsymbol{\ell}_k, \mathcal{B}_k) \in \mathbb{Z}_{\geq 0}^M \times 2^{\mathcal{P}}. \tag{1}$$

For example, if $\boldsymbol{\ell}_k = [0, 2, 1]$, then m_1 is available immediately, m_2 will be available in two time steps, and m_3 in one. The set of available maintainers is

$$\mathcal{A}_k := \{m_j \in \mathcal{M} \mid \ell_j^k = 0\}.$$

3.3 Defender Action Space

At each time step, the defender $\mathcal{D}$ is responsible for assigning a subset of unresolved PRs from the backlog to the currently available maintainers. Each PR may be assigned to at most one maintainer, and each available maintainer may be assigned at most one PR.

Formally, a defender action at time k is a set of assignments

$$d_k \subseteq \mathcal{B}_k \times \mathcal{A}_k,$$

subject to the constraints

$$\begin{aligned} \forall \sigma \in \mathcal{B}_k : \quad & |\{m \mid (\sigma, m) \in d_k\}| \leq 1, \\ \forall m \in \mathcal{A}_k : \quad & |\{\sigma \mid (\sigma, m) \in d_k\}| \leq 1. \end{aligned}$$

These constraints ensure that no maintainer is overloaded and no PR is redundantly reviewed. The defender's full action space at state s_k is then defined as

$$\mathcal{D}(s_k) := \left\{ d \subseteq \mathcal{B}_k \times \mathcal{A}_k \;\middle|\; \begin{array}{l} \forall \sigma \in \mathcal{B}_k : \; |\{m \mid (\sigma, m) \in d\}| \leq 1 \\ \forall m \in \mathcal{A}_k : \; |\{\sigma \mid (\sigma, m) \in d\}| \leq 1 \end{array} \right\}. \tag{2}$$

Each $d_k \in \mathcal{D}(s_k)$ represents a valid assignment decision made by the defender at time step k.

3.4 Attacker Action and Benign PR Arrivals

At every time step k, a new batch of benign PRs arrives exogenously. These PRs are sampled from a probability distribution over subsets of benign PRs

$$\beta_k^{\text{ben}} \sim \mathit{\Pi}, \quad \mathit{\Pi} : 2^{\mathcal{P}_{\text{ben}}} \to [0,1],$$

where $\mathcal{P}_{\text{ben}}$ denotes the set of all possible benign PRs.

In contrast, the attacker $\mathcal{A}$ acts strategically. At each time step k, the attacker may submit a batch of malicious PRs, denoted by $\hat{\beta}_k \subseteq \mathcal{P}_{\text{mal}}$, where $\mathcal{P}_{\text{mal}}$ is the set of all possible malicious PRs. However, to remain stealthy and avoid detection based solely on volume, the attacker must ensure that the total number of PRs in the system (existing backlog plus new malicious PRs) does not exceed a predefined threshold $B_{\max}$, i.e.,

$$|\mathcal{B}_k \cup \hat{\beta}_k| \leq B_{\max}.$$

This constraint prevents the attacker from overwhelming the project with PR flooding, while still allowing room to inject malicious contributions incrementally over time.

3.5 Resolution Model

We let $E \in [0,1]^{3\times 3}$ be the detection probability matrix, where E_{ij} is the probability that a maintainer of expertise level i detects a malicious PR of priority j.

In particular, resolving a PR requires a certain number of time steps based on its priority as well as the expertise level of the assigned maintainer. Let $W = [w_{ij}] \in \mathbb{Z}^{+^{3\times 3}}$, where w_{ij} denotes the time steps needed for a maintainer with level expertise i in resolving a PR with category j.

3.6 Reward Functions

The reward model reflects the success or failure of the defender in managing PRs accurately, and of the attacker in deceiving the system. The defender $\mathcal{D}$ receives a reward for each assignment made at time step k, depending on whether the assigned PR is benign or malicious, and in the latter case, whether the malicious intent is detected by the assigned maintainer.

Let $(\sigma, m) \in d_k$ be a PR-to-maintainer assignment at time step k, where a is the expertise level of maintainer m, and b is the priority level of PR σ. The defender's reward for a single assignment is given by

$$r^{\mathcal{D}}(\sigma, m) = \begin{cases} R_{ab}^{P}, & \text{if } \sigma \text{ is malicious and detected} \\ R_{ab}^{U}, & \text{if } \sigma \text{ is malicious and undetected (merged)} \\ R_{b}^{N}, & \text{if } \sigma \text{ is benign} \end{cases} \tag{3}$$

where R^P_{ab} denotes the reward for successfully detecting a malicious PR, R^U_{ab} is the (negative) utility for missing it, and R^N_b is the utility for correctly processing a benign PR of priority b.

The total reward for the defender at time step k is the sum over all assignments made at that step, i.e.,

$$R^{\mathcal{D}}_k = \begin{cases} \sum_{(\sigma,m)\in d_k} r^{\mathcal{D}}(\sigma, m), & \text{if } |d_k| > 0 \\ R^{\emptyset}, & \text{if } |d_k| = 0\,. \end{cases} \tag{4}$$

Therefore, if no assignments are made at a given time step (i.e., $|d_k| = 0$), the defender receives a fixed reward $R^{\emptyset} < 0$. This term serves as an incentive to avoid idling during the game.

The attacker $\mathcal{A}$ has a different reward structure. Since the attacker's goal is to get malicious PRs accepted undetected, their reward is derived directly from the defender's response to malicious PRs. In particular, we assume the attacker receives a penalty for each malicious PR that is either detected or rejected, and no reward for benign PRs.

Then the attacker's reward for each assignment is defined as:

$$r^{\mathcal{A}}(\sigma, m) = \begin{cases} -R^P_{ab}, & \text{if } \sigma \text{ is malicious and detected} \\ -R^U_{ab}, & \text{if } \sigma \text{ is malicious and undetected (merged)} \\ 0, & \text{if } \sigma \text{ is benign} \end{cases} \tag{5}$$

Thus, the total attacker reward at time step k is given by

$$R^{\mathcal{A}}_k = \sum_{\substack{(\sigma,m)\in d_k: \\ t(\sigma)=\text{malicious}}} r^{\mathcal{A}}(\sigma, m) \tag{6}$$

where $t(\sigma) \in \{\text{benign}, \text{malicious}\}$ is the type of PR σ.

3.7 Transition Dynamics

Given state $s_k = (\boldsymbol{\ell}_k, \mathcal{B}_k)$, and actions d_k, and $\hat{\beta}_k$, the game transitions to the next state

$$s_{k+1} = \mathcal{T}(s_k, d_k, \beta^{\text{ben}}_k, \hat{\beta}_k)\,, \tag{7}$$

with transitions defined as follows:

- For each assignment $(\sigma, m) \in d_k$, set

$$\ell^{k+1}_m = W_{ab}, \text{where } a \text{ is the expertise of } m \text{ and } b \text{ the priority of } \sigma$$

- For other maintainers j, decrement their load as

$$\ell^{k+1}_j = \max(0, \ell^k_j - 1)$$

– Update backlog as

$$\mathcal{B}_{k+1} = (\mathcal{B}_k \setminus \{\sigma \in \mathcal{B}_k \mid \exists m \in \mathcal{A}_k,\ (\sigma, m) \in d_k\}) \cup \beta_k^{\text{ben}} \cup \hat{\beta}_k.$$

Note that we have an asymmetric information structure in which the attacker $\mathcal{A}$ has full knowledge of the current game state as well as the actions taken by the defender. In contrast, the defender $\mathcal{D}$ observes the full state of the system, including the backlog of unresolved PRs and the availability of maintainers, but does not know the type (malicious or benign) of any given PR. That is, $\mathcal{D}$ can see all PRs in the backlog but cannot distinguish between attacker-submitted and benign contributions.

3.8 Problem Objective

Each player in the game seeks to maximize their own cumulative reward over a finite time horizon of length T. The defender $\mathcal{D}$ aims to maximize the total utility gained from correctly assigning and resolving PRs, while minimizing the cost of overlooking malicious ones. The attacker $\mathcal{A}$, on the other hand, attempts to maximize the effectiveness of their malicious PR injections by evading detection and causing harmful PRs to be merged.

Formally, the defender's objective is to choose a sequence of assignment actions $\{d_0, \ldots, d_{T-1}\}$ that maximizes expected cumulative reward over the horizon:

$$\mathcal{D}: \quad \max_{d_0,\ldots,d_{T-1}} \mathbb{E}\left[\sum_{k=0}^{T-1} R_k^{\mathcal{D}}\right] \tag{8}$$

The attacker selects a sequence of malicious PR batches $\{\hat{\beta}_0, \ldots, \hat{\beta}_{T-1}\}$ in order to maximize their own cumulative reward, which reflects the success of undetected malicious contributions, i.e.,

$$\mathcal{A}: \quad \max_{\hat{\beta}_0,\ldots,\hat{\beta}_{T-1}} \mathbb{E}\left[\sum_{k=0}^{T-1} R_k^{\mathcal{A}}\right] \tag{9}$$

4 Methodology

4.1 PPO-Based Agent Training

To train both the attacker and defender agents, we adopt Proximal Policy Optimization (PPO) [22], a widely used online policy gradient method. PPO is known for its ability to perform multiple epochs of minibatch updates while maintaining training stability. Unlike traditional policy gradient methods [14], which perform a single update per sample, PPO uses a clipped surrogate objective to prevent overly aggressive policy changes.

The core idea is to maximize the following total objective function:

$$L_t^{\text{PPO}}(\theta) = \hat{\mathbb{E}}_t\left[L_t^{\text{CLIP}}(\theta) - c_1 L_t^{\text{VF}}(\theta) + c_2 \mathcal{H}\left(\pi_\theta\right)\left(s_t\right)\right], \tag{10}$$

where θ are the policy parameters, c_1 and c_2 are weighting coefficients, and $\hat{\mathbb{E}}_t$ denotes an empirical average over collected timesteps.

The first term $L_t^{\text{CLIP}}(\theta)$ is the clipped surrogate objective that ensures updates do not deviate too far from the previous policy. It is defined as

$$L_t^{\text{CLIP}}(\theta) = \min\left(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right), \tag{11}$$

where the policy ratio $r_t(\theta)$ is given by

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}, \tag{12}$$

and $\hat{A}_t$ is the estimated advantage of action a_t taken in state s_t.

We compute $\hat{A}_t$ using Generalized Advantage Estimation (GAE) [21], which balances bias and variance in the gradient estimate. The advantage is computed as

$$\hat{A}_t = \sum_{l=0}^{T-t} (\gamma\lambda)^l \delta_{t+l}, \quad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t), \tag{13}$$

where δ_t is the one-step temporal difference error, $\gamma \in [0, 1)$ is the discount factor, and $\lambda \in [0, 1]$ controls the bias-variance tradeoff. Here, r_t represents the reward received from the environment. In our formulation (see Section 3), this corresponds to the player-specific reward functions defined for the defender and attacker. Specifically, $r_t = R_k^{\mathcal{D}}$ or $r_t = R_k^{\mathcal{A}}$ as given in (4) and (6), depending on which player is being trained. These rewards are computed from the assignment actions taken by the defender or the malicious PR batches submitted by the attacker.

The second term in the PPO objective is the value function loss, defined as

$$L_t^{\text{VF}}(\theta) = \left(V_\theta(s_t) - V_t^{\text{targ}}\right)^2, \tag{14}$$

where V_t^{targ} is the bootstrap target for the value function. The value function $V_\theta(s_t)$ estimates the expected cumulative reward starting from state s_t under policy π_θ. In our attacker-defender environment, the state s_t corresponds to the game state as defined in Eq. (1). Thus, the value function approximates

$$V(s_t) = \mathbb{E}_{\pi_\theta}\left[\sum_{l=0}^{T-t} \gamma^l r_{t+l} \mid s_t\right],$$

where the transition dynamics are defined by (7).

The third term in the PPO objective is an entropy bonus, which encourages the policy to maintain high entropy and thus supports exploration

$$\mathcal{H}(\pi_\theta)(s_t) = \mathbb{E}_{a_t \sim \pi_\theta}\left[-\log \pi_\theta(a_t|s_t)\right]. \tag{15}$$

Without this term, the policy might prematurely collapse to deterministic behavior, which can harm exploration in large or partially observed action spaces.

In our implementation, we use a clipping threshold of $\epsilon = 0.2$, and we tune the coefficients c_1 and c_2 to balance value estimation and exploration performance. PPO enables stable training in high-dimensional, constrained action spaces such as ours, where not all PR assignments are valid at every timestep.

4.2 PSRO Training Loop

As described in Section 3, we model the adversarial PR problem as a two-player sequential game between the defender $\mathcal{D}$ and attacker $\mathcal{A}$. To learn effective strategies in this setting, we adopt a multi-agent reinforcement learning (MARL) framework based on Policy-Space Response Oracles (PSRO) [11].

One of the central challenges in MARL is the non-stationarity that arises when multiple agents update their policies simultaneously. A common baseline is Independent Reinforcement Learning (InRL), where each agent treats other agents as part of the environment. However, this often leads to brittle or non-generalizable policies, particularly in adversarial or partially observable settings.

To overcome this, we use the PSRO framework, which generalizes InRL, iterated best response, and double oracle methods. In PSRO, each agent maintains a population of policies and iteratively trains new policies as approximate best responses to mixtures over the opponent's population. Meta-strategies are computed using empirical game-theoretic analysis (EGTA) on a meta-game: a normal-form matrix game where each pure strategy is a full policy from the population, and payoffs are estimated via pairwise simulations using the cumulative rewards $R_k^{\mathcal{D}}$ and $R_k^{\mathcal{A}}$.

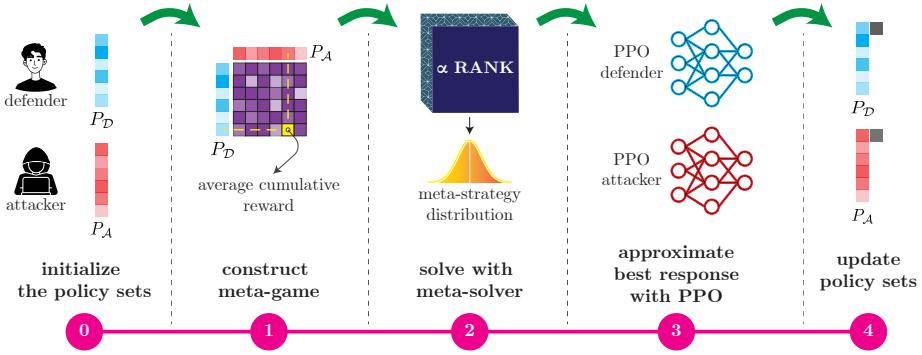

Fig. 3: PSRO training loop

As illustrated in Figure 3, the PSRO loop proceeds as follows. Initially, both $\mathcal{D}$ and $\mathcal{A}$ have one policy each (e.g., uniformly random). At each outer-loop iteration:

1. A meta-game is constructed where rows and columns correspond to policies in policy sets $P_{\mathcal{D}}$ and $P_{\mathcal{A}}$, respectively. The payoff for each entry is the

average cumulative reward obtained by running the corresponding pair of defender and attacker policies.
2. A meta-solver, such as α-Rank [15], is used to compute a meta-strategy (a distribution over policies) for each player. This distribution captures how often each policy would be selected under evolutionary dynamics.
3. Each player trains a new policy via PPO (see Eq. (10)) as an approximate best response to the opponent's meta-strategy. The training is done in the same game environment described earlier, using the reward functions and transition dynamics defined in our model.
4. The newly trained policy is added to the agent's population.

This process is repeated for K iterations. Over time, the policy populations grow, and the meta-strategies evolve to reflect a richer set of strategic behaviors, which improves generalization and robustness. The complete PSRO training loop with PPO oracles is shown in Algorithm 1.

Algorithm 1: PSRO Training with PPO as Oracle

Input: Initial policy sets $P_{\mathcal{D}}, P_{\mathcal{A}}$; meta-solver μ; iterations K; PPO parameters
for $k = 1$ *to* K **do**
  Step 1: Solve Meta-Game
  Compute meta-strategies $(\psi_{\mathcal{D}}, \psi_{\mathcal{A}}) \leftarrow \mu(P_{\mathcal{D}}, P_{\mathcal{A}})$
  for *player* $\in \{\mathcal{D}, \mathcal{A}\}$ **do**
    Step 2: Train Best Response via PPO
    Initialize policy $\pi_\theta^{\text{player}}$ with random parameters
    for *PPO update* $= 1$ *to* U **do**
      Collect trajectories:
      for *episode* $= 1$ *to* $N_{episodes}$ **do**
        Sample opponent policy $\pi_{\text{opp}} \sim \psi_{\text{opp}}$
        Roll out interaction between $\pi_\theta^{\text{player}}$ and π_{opp} for T steps
        Store (s_t, a_t, r_t, s_{t+1}) in buffer
      Compute advantage estimates $\hat{A}_t$ and returns $\hat{R}_t$ using GAE
      Compute PPO objective $L_t^{\text{PPO}}(\theta)$ (10) and update policy parameters θ
    Add trained policy $\pi_\theta^{\text{player}}$ to P_{player}

5 Experiments

5.1 Setup

To evaluate our solution methodology, we consider two real-world open-source projects: Pandapower [23], which is an open source tool for power system modeling, analysis and optimization, and NumPy [8] which is a fundamental package for scientific computing with Python. As of June 2025, Pandapower and

NumPy have been downloaded over 1 million and 11 billion times, respectively, which highlights their widespread adoption and large downstream user base. We chose these two projects for our case studies to evaluate the effectiveness of our methodology in both small and large environments. We have extracted the PR and maintainer information for these two packages using a custom tool we have developed for this purpose.[3] The data was extracted over a two-week period in June 2025. Based on the collected data for both projects, the parameters used in our experiments are summarized in Table 1.

Category	**Parameter**	**Value**
`Global`	T	100
	E	$\begin{bmatrix} 0.57 & 0.35 & 0.10 \\ 0.79 & 0.50 & 0.25 \\ 0.89 & 0.55 & 0.40 \end{bmatrix}$
	W	$\begin{bmatrix} 3 & 3 & 3 \\ 2 & 2 & 2 \\ 1 & 1 & 1 \end{bmatrix}$
	R^P	$[60.0, 24.0, 12.0]$
	R^U	$[-20.0, -12.0, -4.0]$
	R^N	$[2.6, 1.2, 0.4]$
	$R^\emptyset$	-5.0
`small`	$\lvert\mathcal{M}_{\text{Senior}}\rvert$	1
	$\lvert\mathcal{M}_{\text{Intermediate}}\rvert$	0
	$\lvert\mathcal{M}_{\text{Junior}}\rvert$	2
	$\lvert\mathcal{S}\rvert$	432
	$B_{\max}$	$[2, 2, 2]$
`large`	$\lvert\mathcal{M}_{\text{Senior}}\rvert$	2
	$\lvert\mathcal{M}_{\text{Intermediate}}\rvert$	3
	$\lvert\mathcal{M}_{\text{Junior}}\rvert$	3
	$\lvert\mathcal{S}\rvert$	746,494
	$B_{\max}$	$[5, 5, 5]$

Table 1: Parameters used to initialize defender and attacker models

[3] `https://github.com/j0m0k0/PuReX`

For both experiments, training is conducted as described in Section 4. Each agent is trained for a total of 100,000 timesteps in the small environment and 500,000 timesteps in the large environment, including both hyperparameter tuning and policy learning. All training was performed on an Apple machine with an M4 chip with 10 CPU cores and 10 GPU cores. Training took roughly 3 and 12 hours for the small and large environments, respectively.

5.2 Baseline Policies

To evaluate our trained defender and attacker agents, we introduce two baseline policies: `Random` and `Greedy`:

- Under the `Random` policy, the defender assigns the PRs randomly to the available maintainers based on a uniform probability distribution until reaching the end of the horizon T. For the attacker, the `Random` policy injects malicious PRs uniformly at random across the PR types at each time step, without considering the state of the backlog or the behavior of the defender.
- In the `Greedy` policy, the defender prioritizes maintainers based on their level of expertise. That is, she first assigns PRs to all available *Senior* maintainers, followed by *Intermediates*, and finally to *Juniors*. On the attacker side, the `Greedy` strategy injects the maximum number of malicious PRs allowed at each step, targeting PR types that are least likely to be detected, i.e., those with the highest PR category first, then the second highest priority and so on.

5.3 Defender Performance

We used **Algorithm 1** to train both players for five iterations in both small and large environments. Figures 4 and 5 show the training loss for $\mathcal{D}$ in small and large environments, respectively. One can observe more consistent training behavior in the small environment compared to the large one, due to the significantly larger action space in the latter, which makes convergence more challenging. In both environments, $\mathcal{D}$ is able to successfully optimize the parameters of the underlying neural network used by PPO, effectively minimizing its training loss.

The entropy loss (i.e., the negative of entropy), depicted in Figures 6 and 7, indicates that, over the course of training, $\mathcal{D}$ becomes more certain about her actions, and the entropy loss plateaus near zero. Comparing the entropy loss across training iterations shows that the defender is initially more confident, resulting in lower entropy. However, as training progresses and the attacker adopts more sophisticated tactics, the defender's policy becomes more uncertain, leading to a gradual increase in entropy (i.e., lower entropy loss).

We compare our trained defender policies, referred to as PRD, against random and greedy baselines introduced in Section 5.2. The results are summarized in Table 2. Our trained defender consistently outperforms the baselines, achieving higher average rewards, a greater percentage of detected malicious PRs, and

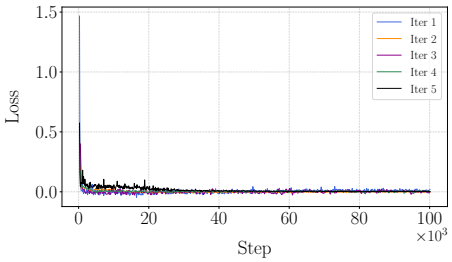

Fig. 4: Training loss for $\mathcal{D}$ (small)

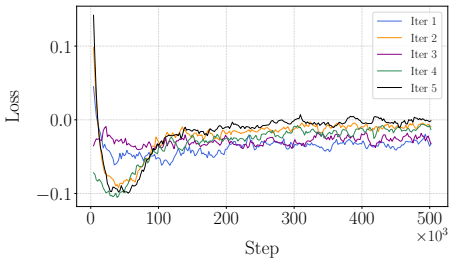

Fig. 5: Training loss for $\mathcal{D}$ (large)

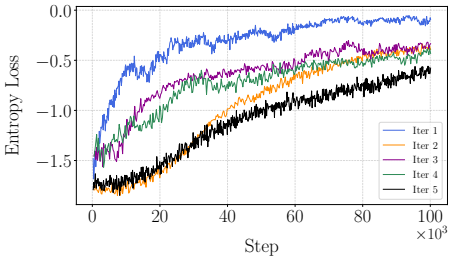

Fig. 6: Entropy loss for $\mathcal{D}$ (small)

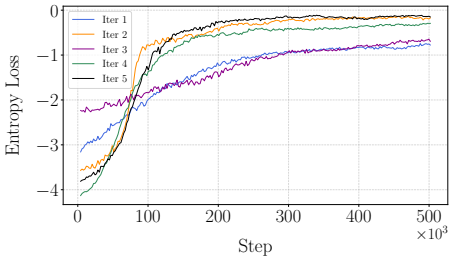

Fig. 7: Entropy loss for $\mathcal{D}$ (large)

lower average backlog size, demonstrating strong generalization performance. The results are averaged over 1,000 independent runs, each consisting of 100 uniformly sampled episodes. In the large environment, the evaluation is conducted over 1,000 episodes.

One of the main concerns in MARL is generalizability. Owning to the solid PSRO framework that we use for agent training and the PPO algorithm for policy approximation, our agents generalize well in comparison to the proposed baselines. The results for our trained defender agent (PRD) are shown in Figure 8. As shown the policy of the defender (PRD-v5) outperforms all other defense policies against all types of attackers.[4] The attack policies are indicated in the titles of the sub-plots.

5.4 Attacker Performance

The training loss for the attacker agent is shown in Figures 9 and 10, corresponding to the small and large environments, respectively. Training is slightly more stable and successful in the small environment. In contrast, the attacker encounters greater difficulty during the initial iterations in the large environment, but performance improves as training progresses. The level of uncertainty of the attacker is measured by the entropy loss and is shown in Figures 11 and

[4] We use PRD-vx (PRA-vx) to refer to our trained defender (attacker) at the end of iteration x.

Env	Policy of $\mathcal{D}$	Avg. $R^{\mathcal{D}}$	Avg. $R^{\mathcal{A}}$	Mal. PRs Detected	Avg. Backlog
`small`	**PRD-v5**	16.2 ± 0.8	-25.7 ± 0.6	$84.3\% \pm 2.4\%$	1.9 ± 0.4
`small`	Greedy	11.5 ± 1.3	-17.0 ± 1.6	$66.1\% \pm 3.8\%$	2.8 ± 0.6
`small`	Random	5.8 ± 2.1	-4.7 ± 1.0	$41.6\% \pm 5.6\%$	4.6 ± 1.0
`large`	**PRD-v5**	14.8 ± 1.0	-23.5 ± 1.4	$80.2\% \pm 2.9\%$	2.3 ± 0.5
`large`	Greedy	10.4 ± 1.5	-15.0 ± 0.7	$62.4\% \pm 3.9\%$	3.1 ± 0.7
`large`	Random	5.3 ± 2.4	-3.3 ± 1.3	$39.1\% \pm 5.2\%$	4.9 ± 1.2

Table 2: Defender agent performance comparison against PRA-v4.

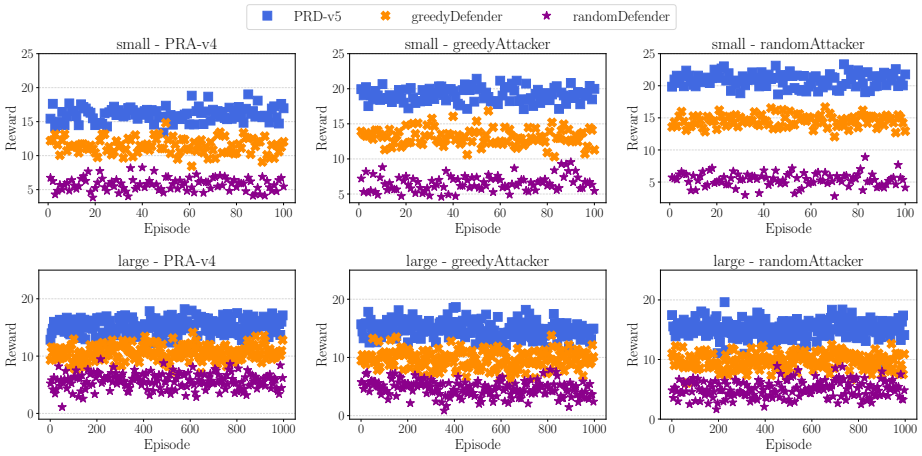

Fig. 8: Defender rewards: Comparison of 3 different defenders (PRD-v5, Greedy, and Random) against different fixed attackers in both small (top) and large (bottom) environments.

12. The results show that the attacker becomes more confident over time, but in the large environment, the attacker still remains uncertain at choosing optimal actions, even after half a million training steps.

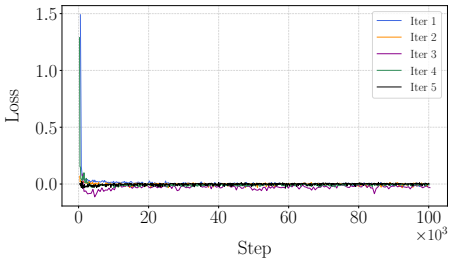

Fig. 9: Training loss for $\mathcal{A}$ (small)

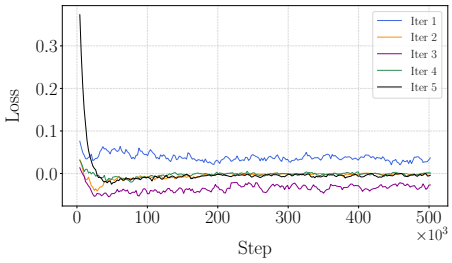

Fig. 10: Training loss for $\mathcal{A}$ (large)

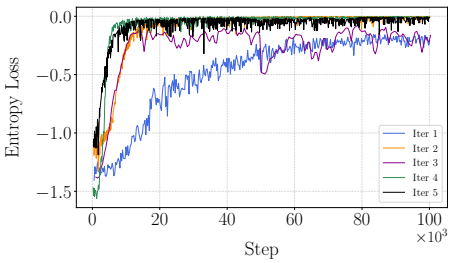

Fig. 11: Entropy loss for $\mathcal{A}$ (small)

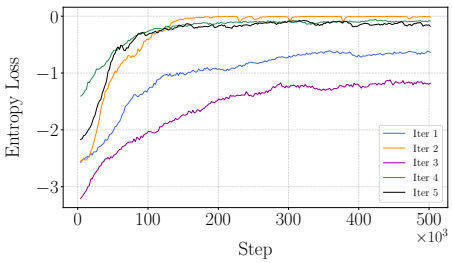

Fig. 12: Entropy loss for $\mathcal{A}$ (large)

We also compare the performance of our trained attacker with random and greedy baselines. The results of these comparisons are shown in Table 3. The results show how different attacker policies (including our trained policy, PRA), perform against the 4th iteration of our trained defender. As we can see, PRA outperforms the baseline against the trained defender PRD-v4. To show the generalizability of our model, we also show a pairwise comparison of various attackers against various fixed defenders in Figure 13. As shown, the trained attacker agent (PRA-v5) consistently outperforms the baselines against all types of defenders. The strategy of the defender is indicated in the title of the corresponding sub-plot.

Env	Policy of $\mathcal{A}$	Avg. $R^{\mathcal{A}}$	Avg. $R^{\mathcal{D}}$	Mal. PRs Injected
`small`	**PRA-v5** (ours)	-7.6 ± 1.1	8.2 ± 1.0	$47.6\% \pm 2.8\%$
`small`	Greedy	-18.5 ± 1.3	13.4 ± 1.1	$32.9\% \pm 3.5\%$
`small`	Random	-23.4 ± 1.5	14.1 ± 1.4	$20.4\% \pm 4.2\%$
`large`	**PRA-v5** (ours)	-15.2 ± 1.1	9.8 ± 1.1	$41.8\% \pm 3.1\%$
`large`	Greedy	-19.4 ± 1.4	13.2 ± 1.2	$29.5\% \pm 4.0\%$
`large`	Random	-22.1 ± 1.6	14.7 ± 1.4	$22.1\% \pm 5.1\%$

Table 3: Attacker agent performance comparison against PRD-v4.

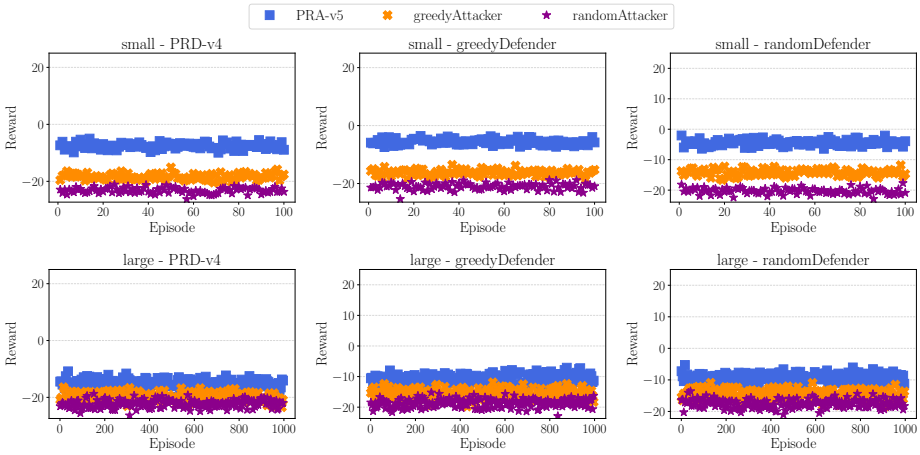

Fig. 13: Attacker rewards: Comparison of 3 different attackers (PRA-v5, Greedy, and Random) against different fixed defenders in both small (top) and large (bottom) environments.

6 Conclusion

Protecting OSS projects demands strategic PR assignment to maintainers, especially in the presence of stealthy attackers who attempt to inject malicious PRs. To our knowledge, this work is the first to formalize the PR assignment problem in adversarial environments, using a game-theoretic framework. We present a novel two-player, general-sum, partially-observable game model that captures key features that can be gleaned from a public pull-based software development environment. While a single undetected malicious PR could be catastrophic in practice, our focus is on modeling stealthy, persistent attackers who may introduce malicious code incrementally over time rather than in a single PR. This allows us to capture a broader range of realistic attack strategies and to evaluate defensive policies under sustained adversarial pressure, rather than terminating the game after the first success. Through the use of the PSRO framework, we derive effective defense strategies that outperform baseline PR assignment policies and identify strong attacker strategies. To demonstrate the behavior of the policies obtained, information from two popular Python OSS packages are used to instantiate the model. Our evaluation results show the robustness and the generalization of the policies obtained.

Acknowledgment

This work was supported in part by NSF Award CCF-2106339.

References

1. Node.js package manager, `https://www.npmjs.com/`, [Accessed 2025-06-23].
2. The python package index, `https://pypi.org/`, [Accessed 2025-06-23].
3. Azeem, M.I., Peng, Q., Wang, Q.: Pull request prioritization algorithm based on acceptance and response probability. In: 2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS). pp. 231–242. IEEE (2020)
4. BlackDuck: 2025 Open Source Security and Risk Analysis Report. BlackDuck (2025), `https://www.blackduck.com/content/dam/black-duck/en-us/reports/rep-ossra.pdf`, [Accessed 2025-06-23].
5. Boucher, N., Anderson, R.: Trojan source: Invisible vulnerabilities. In: 32nd USENIX security symposium (USENIX Security 23). pp. 6507–6524 (2023)
6. Chacon, S., Straub, B.: Pro git. Springer Nature (2014)
7. Gousios, G., Pinzger, M., Deursen, A.v.: An exploratory study of the pull-based software development model. In: Proceedings of the 36th international conference on software engineering. pp. 345–355 (2014)
8. Harris, C.R., Millman, K.J., Van Der Walt, S.J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N.J., et al.: Array programming with numpy. Nature **585**(7825), 357–362 (2020)
9. Khatoonabadi, S., Abdellatif, A., Costa, D.E., Shihab, E.: Predicting the first response latency of maintainers and contributors in pull requests. IEEE Transactions on Software Engineering (2024)

10. Ladisa, P., Plate, H., Martinez, M., Barais, O.: Sok: Taxonomy of attacks on open-source software supply chains. In: 2023 IEEE Symposium on Security and Privacy (SP). pp. 1509–1526. IEEE (2023)
11. Lanctot, M., Zambaldi, V., Gruslys, A., Lazaridou, A., Tuyls, K., Pérolat, J., Silver, D., Graepel, T.: A unified game-theoretic approach to multiagent reinforcement learning. Advances in neural information processing systems **30** (2017)
12. Lawson, A.: 2024 Global Spotlight Insights Report. The Linux Foundation (December 2024)
13. de Lima Júnior, M.L., Soares, D.M., Plastino, A., Murta, L.: Developers assignment for analyzing pull requests. In: Proceedings of the 30th annual ACM symposium on applied computing. pp. 1567–1572 (2015)
14. Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., Kavukcuoglu, K.: Asynchronous methods for deep reinforcement learning. In: International conference on machine learning. pp. 1928–1937. PmLR (2016)
15. Muller, P., Omidshafiei, S., Rowland, M., Tuyls, K., Perolat, J., Liu, S., Hennes, D., Marris, L., Lanctot, M., Hughes, E., et al.: A generalized training approach for multiagent learning. arXiv preprint arXiv:1909.12823 (2019)
16. Ohm, M., Plate, H., Sykosch, A., Meier, M.: Backstabber's knife collection: A review of open source software supply chain attacks. In: Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17. pp. 23–43. Springer (2020)
17. Page, C.: Apple iCloud, Twitter and Minecraft vulnerable to 'ubiquitous' zero-day flaw. Tech Crunch (2021), https://techcrunch.com/2021/12/10/apple-icloud--twitter-and-minecraft-vulnerable-to-ubiquitous-zero-day-exploit/
18. Pearce, J.M.: Economic savings for scientific free and open source technology: A review. HardwareX **8** (2020)
19. Raywood, D.: Vulnerable instances of log4j still being used nearly 3 years later (Oct 2024), https://www.scworld.com/news/vulnerable-instances-of-log4j-still-being-used-nearly-3-years-later
20. Rong, G., Zhang, Y., Yang, L., Zhang, F., Kuang, H., Zhang, H.: Modeling review history for reviewer recommendation: A hypergraph approach. In: Proceedings of the 44th international conference on software engineering. pp. 1381–1392 (2022)
21. Schulman, J., Moritz, P., Levine, S., Jordan, M., Abbeel, P.: High-dimensional continuous control using generalized advantage estimation. arXiv preprint arXiv:1506.02438 (2015)
22. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)
23. Thurner, L., Scheidler, A., Schäfer, F., Menke, J.H., Dollichon, J., Meier, F., Meinecke, S., Braun, M.: Pandapower—an open-source python tool for convenient modeling, analysis, and optimization of electric power systems. IEEE Transactions on Power Systems **33**(6), 6510–6521 (2018). https://doi.org/10.1109/TPWRS.2018.2829021
24. Wu, Q., Lu, K.: On the feasibility of stealthily introducing vulnerabilities in open-source software via hypocrite commits. Proc. Oakland **17** (2021)
25. Ye, X., Zheng, Y., Aljedaani, W., Mkaouer, M.W.: Recommending pull request reviewers based on code changes. Soft Computing **25**, 5619–5632 (2021)
26. Zimmermann, M., Staicu, C.A., Tenny, C., Pradel, M.: Small world with high risks: A study of security threats in the npm ecosystem. In: 28th USENIX security symposium (USENIX Security 19). pp. 995–1010 (2019)