Changing Nothing, Yet Changing Everything: Exploring Rug Pulls in GitHub Workflows

Edoardo Riggio
Software Institute, *USI* Lugano, Switzerland
edoardo.riggio@usi.ch

Cesare Pautasso
Software Institute, *USI* Lugano, Switzerland
c.pautasso@ieee.org

***Abstract*—Software supply chain attacks have become a significant threat to modern software systems. By exploiting the complex and transitive nature of dependencies, malicious actors have been able to perform significant attacks, also taking advantage of the dynamic relationship between software components and their dependencies. In Continuous Integration and Continuous Deployment (CI/CD) ecosystems such as GitHub Actions, developers assemble workflows out of reusable Actions. However, these Actions—in particular JavaScript ones—come with an intricate network of dependencies. As they evolve, these dependency networks expose GitHub CI/CD pipelines to subtle vulnerabilities that may be introduced without any modification of the workflows themselves. This paper investigates such phenomenon, which we call "rug pull" within GitHub workflows. Through formalization and an empirical analysis of real-world workflows, we characterize the propagation and persistence of such vulnerabilities as well as their remediation. Our findings highlight architectural considerations needed when designing secure yet maintainable CI/CD pipelines, emphasizing the need for careful dependency management and coordinated responsibility across the software supply chain.**

***Index Terms*—DevSecOps, GitHub Actions, software supply chain, security vulnerability**

I. Introduction

Reusability is a fundamental principle in software architecture, enabling developers to build software systems by leveraging existing components. This practice has led to the emergence of software supply chains (SSC), where third-party components are mixed in with first-party ones. Relying on third-party components leads to the formation of complex dependency networks in which dependencies are maintained and evolve independently over time [1, 2].

GitHub Actions [3] exemplify this paradigm as composable and reusable building blocks of the GitHub Continuous Integration and Continuous Deployment (CI/CD) ecosystem. Each Action, however, rarely lives in isolation. Actions can use, for example, JavaScript and Docker dependencies. Such dependencies can also become targets of supply chain attacks. The dependencies, by being compromised, can affect every GitHub workflow relying on them.

This paper focuses specifically on JavaScript (JS) GitHub Actions. We examined how vulnerabilities in JS packages propagate and affect the security posture of GitHub Actions and workflows themselves. The transitive nature of dependencies, paired with the flexible binding of workflows to mutable GitHub Actions versions, results in vulnerabilities which can lurk undetected for extended periods of time, even if the workflow code itself remains unchanged. Dynamic binding of Actions into workflows has clear maintenance benefits, as workflows automatically run with the latest revision of the Action version they depend on. However, this introduces important security issues, as highlighted by the March 2025 compromise of the widely used `tj-actions/changed-files` GitHub Action [4]. This supply chain attack leveraged exactly this mutability of GitHub Actions versions to inject malicious code without altering the workflow specification.

Our research aims to provide a empirical and rigorous characterization of these supply chain attacks, which we called "rug pulls." Workflow developers choose a specific Action version, trusting that it will not be maliciously modified underneath the workflow. In particular, we will focus on the history of the workflows themselves, by analyzing the temporal dynamics of the detection and remediation of such rug pulls. By shedding light on this issue, we aim to help DevSecOps architects and practitioners to design not only efficient but also more resilient, secure, and maintainable CI/CD pipelines that leverage reusable components without compromising security.

For this empirical study, we rely on a dataset composed of 1 026 open source repositories from GitHub, 3 543 GitHub workflows, and 60 287 workflow commits distributed from August 2019 to November 2025 to answer the following research questions:

RQ1: How frequent are rug pulls in JavaScript GitHub Actions? As an initial step, we want to understand to which extent are rug pulls present in our dataset. We provide empirical evidence of a non negligible number of rug pulls present in workflow commits. ***Main Finding:*** 102 Actions (32% of the Actions in our dataset) cause at least one rug pull in 1135 workflows (32% of workflows in our dataset). ***Implication:*** Rug pulls are a recurring phenomenon rather than rare edge cases. Thus, rug pulls must be treated as a primary SSC risk.

RQ2: Which dependencies occur the most in rug pulls? By looking at all the detected rug pulls, we study more in depth the Actions and JS dependencies causing these rug pulls. We show that there are a few Actions and JS dependencies that are used in the majority of cases. We also show the severity of the vulnerability introduced by the JS dependencies. ***Main Finding:*** `actions/checkout` is the most occurring Action in rug pulls (used in 2 235 rug pulls), while

`brace-expansion` is the most occurring JS dependency (used 3 475 times) presenting the vulnerability at the root cause of the rug pull. ***Implication:*** A small set of highly reused JS packages act as SSC "choke points." Hardening and monitoring these components can drastically reduce the exposure of these dependencies to rug pulls.

RQ3: What role do the different maintainers play in the remediation or persistence of rug pulls? We classify rug pull events based on the maintainers (i.e., workflow, Action, and JS dependency maintainers) involved. Moreover, we show how each maintainer is responsible for the remediation or persistence of a rug pull. Finally, we show how coordinated responsibility across the entire ecosystem is needed for the correct and timely remediation to such rug pulls. ***Main Finding:*** In 478 (7%) rug pulls, there is a lack of coordination between the actors involved in the rug pull, where although fixes are made available by one actor, they are never implemented by the other maintainers. ***Implication:*** The improvement of inter-actor notification and streamlined update flows could greatly reduce the number of rug pulls events.

RQ4: What are the mean times to fix or potentially fix a rug pull? How do the rug pulls' ages vary over time? To study the dynamics of the rug pull life cycle, we define different temporal metrics. If the rug pull was fixed, we will see how long it took the workflow or Action maintainer to implement this fix. However if a fix to a rug pull was not detected, we will measure how much time it took to have at least a potential fix or the mean time a vulnerability remained open. We study the trends of these metrics throughout the years. ***Main Finding:*** Rug pulls are often long-lived, as they remain unfixed for hundreds of days on average. Fixes by Action maintainers often arrive quicker than those by workflow maintainers. All the rug pulls introduced in 2019 have been fixed, with an average age of over 5.4 years. ***Implication:*** The observed long remediation times imply that workflows remain vulnerable for long periods of time, underscoring the need for automation and policies that reduce TTX in CI/CD pipelines.

The rest of this paper is structured as follows. In Section II we introduce the concepts of GitHub workflows, Actions, and Action pinning. In Section III we formally define what a rug pull is, how it is fixed and all the temporal metrics used in this study. In Section IV we explain how the dataset used in this study was obtained. In Section V we explain how rug pulls are detected in GitHub workflow commits. In Section VI we answer to the above research questions by applying our methodology to our dataset. Before drawing some conclusion in Section X and presenting the related work in Section IX, we discuss the results in Section VII and present possible threats to validity in Section VIII.

II. Background

GitHub *workflows* are Y(A)ML files that define a CI/CD process. They are saved in the `.github/workflows` directory of a repository. Workflows are composed of triggers, jobs, and steps. Jobs define a series of steps that will be run by a GitHub runner machine.

Workflow steps can run their own custom shell scripts, other workflows, Docker containers, or use pre-made packages called GitHub *Actions*. These actions are developed by the community and can be released on the GitHub Marketplace. Actions can be of three different types, namely JS, Docker, and Composite Actions [5]. Since the majority of actions in our dataset are JS actions, we will focus on them in this study.

When using a specific version of a third-party GitHub action in a workflow, several *pinning methods* are available. One could use a branch or tag name (e.g. `main`), a major version tag (e.g. `v2`), a complete version tag (e.g. `v1.2.0`), or the full-length commit SHA (e.g. `f4c4...25d4`). The first three cases (i.e., branch/tag, major, and complete), however, do not point to an immutable release. This means that the action maintainers could change the code under a specific version tag and affect all the workflows using that tag. As GitHub specifies in its documentation about the secure use of third party actions, "[p]inning an action to a full-length commit SHA is currently the only way to use an action as an immutable release." [6] Moreover, previous research classified not using full-length commit SHAs as a security misconfiguration, and the most occurring one [1, 7–9]. In this paper, we aim to observe when this misconfiguration has turned into a vulnerability in real-world CI/CD workflows.

III. Defining Rug Pulls

A rug pull is an event that happens in the history of a workflow (i.e., in a commit), whenever an action used in a workflow introduces some vulnerabilities without changing its version tag (Fig. 1, ❶). Workflows can refer to a certain version tag of an action (e.g., `v3`). However, the source code of the action with this tag can be modified at any time by the action maintainer. This means that new vulnerabilities can be introduced (or removed) in the SSC of the CI/CD pipeline without any change in the workflow itself.

Finding Rug Pulls

Let W denote a workflow, which uses a set of Actions, each with its own version v. Let c denote a git commit of W happening at time t. Let $D(v, c)$ be the set of dependencies (direct and indirect) that are needed to run a given c of Action version v. Let $D_r(v, c) \subseteq D(v, c)$ be the set of dependencies for which a vulnerability exists in c at time t.

> A rug pull event happens in commit c_r at time $t_r > t_0$ iff W continues to use the same v introduced in commit c_0 at time t_0, and in commit c_r at time t_r a new vulnerable dependency is introduced such that $D_r(v, c_r) \setminus D_r(v, c_r - 1) \neq \varnothing$.

For each identified rug pull, we save t_r, the commit hash of c, its location (i.e. repository and workflow names), v (it can be a major version, e.g., v4; a complete version, e.g., v1.0.2; or a branch/tag version, e.g., main), and $D_r(v)$—which represents a list of vulnerable dependencies that the rug pull introduced.

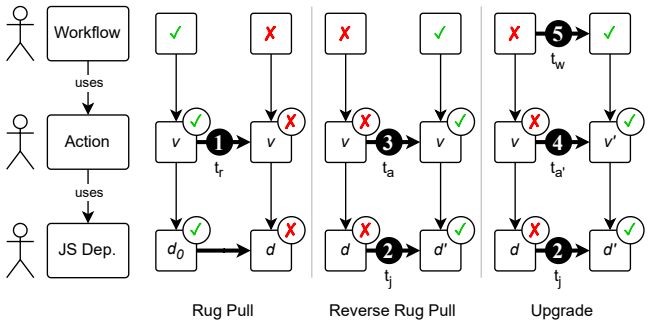

Fig. 1. The three actors involved in a rug pull and the changes they apply to modify dependencies, Actions and workflows. Vulnerable dependencies d are replaced with d', a fixed version of the same dependency or an alternative non-vulnerable dependency.

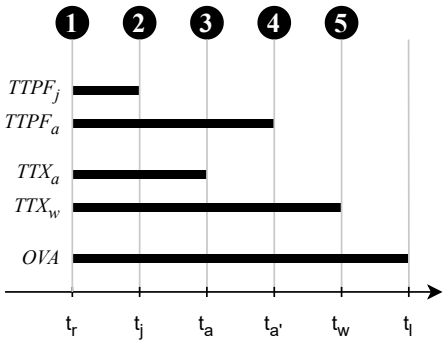

Fig. 2. Temporal metrics related to the rug pull life cycle events.

Fixing Rug Pulls

Through the history of a workflow, a rug pull can be fixed. This can happen through a potential or an actual fix. In general, a fix is detected whenever all the vulnerable dependencies introduced by the rug pull are themselves fixed or not present anymore. The fix can be introduced by two of the three different Actors (as shown in Fig. 1, where ✓ means that the fix was implemented, and ✘ means that it was not).

- **Workflow Maintainer** (Fig. 1, ❺): In commit c_w at time $t_w > t_r$, a workflow maintainer can introduce in w a fix to the rug pull. The fix may happen either by selecting a v' with $D_r(v', c_w) = \varnothing$, or by no longer using Action v in the workflow ($v \notin W$).
- **Action Maintainer** (Fig. 1, ❸): In commit c_a at time $t_a > t_r$, an Action maintainer can change the commit to which v points to so that $D_r(v, c_a) = \varnothing$ once again. Since this fix is obtained without any modification to the workflow, we call it a *reverse rug pull*.

Unfixed and Unfixable Rug Pulls

When none of the previous fix events happen, we have an unfixed rug pull. Whenever a fix is not detected in one of the commits of the workflow, we check if the rug pull is fixable. A rug pull is said to be fixable whenever a fix exists but has not been adopted by either the maintainer of the workflow or of the action. Based on the actor making the fix available, we have two different types of potential fixes:

- **Action Maintainer** (Fig. 1, ❹): In commit $c_{a'}$ at time $t_{a'} > t_r$, an Action maintainer can upgrade v to v', where $D_r(v', c_{a'}) = \varnothing$. This fixed Action version v', however, is never (or not yet, $t_w > t_{a'}$) used by the workflow maintainer.
- **Dependency Maintainer** (Fig. 1, ❷): In commit c_j at time $t_j > t_r$, a dependency maintainer can upgrade $d \in D_r$ to $d' \notin D_r$. This fixed dependency d', however, is never (or not yet, $t_a > t_j$) used by the action maintainer.

Finally, given t_l to be the time of c_l, the last commit of W, if none of c_w or c_a happen before t_l, then we say that the rug pull is unfixable. In this final case, it is the responsibility of the JS dependency maintainer to fix the vulnerability and push a clean version of the dependency so that the Action depending on it may upgrade its dependencies.

Temporal Metrics

To characterize the vulnerability management dynamics of the three actors, we define five temporal metrics (Fig. 2).

Time to Fix (TTX): This metric is similar to the Time to Remediate (TTR) used in vulnerability management [10]. It measures the time it takes for an identified vulnerability to be remediated. Since, in our case, we have no way to know when the vulnerability was first identified, *TTX* computes the time it takes for a rug pull (t_r) to be fixed by the workflow maintainer (t_w) or by the Action maintainer (t_a):

$$TTX_w = t_w - t_r \tag{1}$$

$$TTX_a = t_a - t_r \tag{2}$$

Time to Potential Fix (TTPF): This metric measures the time before which a fix to a rug pull becomes available, either as a consequence of a fix being released for a vulnerable dependency (t_j) or a new, fixed version of an Action becoming available ($t_{a'}$):

$$TTPF_j = t_j - t_r \tag{3}$$

$$TTPF_a = t_{a'} - t_r \tag{4}$$

Open Vulnerability Age (OVA): This metric is also a slight variation of an existing one, commonly used in vulnerability management [10]. Because of the above limitations, we cannot account for vulnerability detection, thus we compute the age of unfixed rug pulls from when they first occurred, relative to the last commit timestamp (t_l):

$$OVA_j = OVA_a = OVA_u = t_l - t_r \tag{5}$$

Here, OVA_j is measured on rug pulls due to dependencies already fixed by by JS dependencies maintainers, but not yet by the corresponding Action maintainer; OVA_a refers to rug pulls for which a new, fixed Action version has been released, but still need to be imported by workflow maintainers; OVA_u refers to unfixable rug pulls, assuming that until the last commit at t_l no potential fix has become available.

TABLE I
DATASET STRUCTURE AND SIZE STATISTICS

Element	Min	Max	Avg	Med	Unique	Total
Repositories	—	—	—	—	—	1027
Workflows[1]	1	62	3.45	2	—	3 543
Commits[2]	0	416	17.02	6	—	60 287
Vulnerabilities[3]	0	76	19.37	16	292	1 168 000
Actions[3]	0	75	5.15	3	321	310 477
Action Versions					2 026	
Vulnerabilities[4]	0	72	6.87	2	292	1 168 000
Dependencies[4]	1	928	202.62	87	4 112	34 449 060
Dep. Versions					20 595	
Vulnerabilities[5]	0	14	0.03	0	292	1 168 000

[1]Per repository [2]Per workflow [3]Per commit [4]Per Action [5]Per dependency

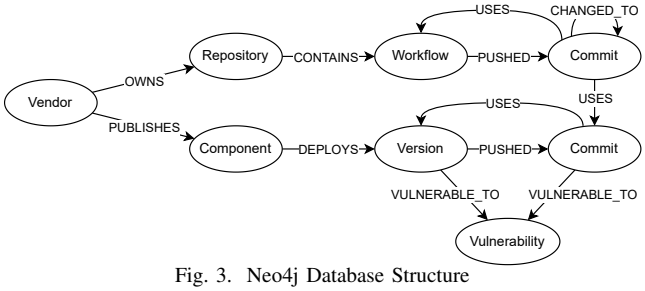

Fig. 3. Neo4j Database Structure

IV. DATASET

For this study, we decided to take the 1027 most starred repositories on GitHub [11]. To be used, each repository must contain at least one workflow in its `.github/workflow` directory. The artifacts commit timestamps span from August 15, 2019 to November 7, 2025. To collect the dataset, we built a tool called *Kleio*. *Kleio* is a crawler written in Golang, that, given a repository, will retrieve the whole history of every workflow file found in it. In addition, all the direct and indirect dependencies of the workflows are scraped. The data is pre-processed and then saved in a Neo4j database. Table I shows some statistics about the dataset content. Figure 3 shows the full structure (node and relationship types) of the Neo4j database used by *Kleio*. Both *Kleio* and the dataset used in this study are available in the replication package [12].

GitHub Workflow History: To track the entire history of workflows, in the database *Workflow* nodes are connected to *Commit* nodes (Figure 3). Each pair of subsequent commits is compared and a diff is stored in the *CHANGED_TO* relationship. This diff is computed by a slightly modified version of the GAWD tool developed by Mazrae et al. [13] As in this study we do not consider custom shell scripts or Docker containers, but focus on Actions, Workflow *Commit* nodes can be connected to one or more *Workflow* or component *Commit* nodes. In this case, a component *Commit* node represents a GitHub Action used by the Workflow, whose versions and the corresponding commits are also tracked. All the nodes connected to the Workflow *Commit* through a *USES* relationship, which represents a direct dependency of the workflow.

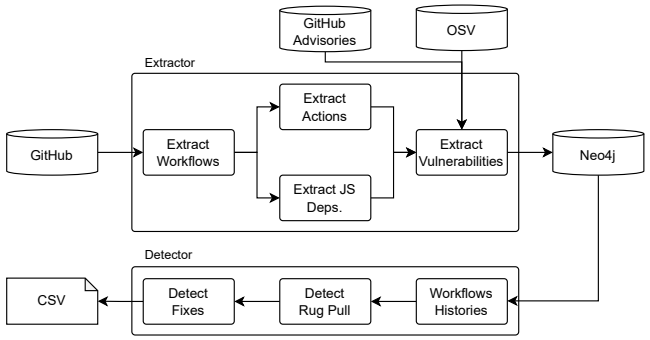

Fig. 4. *Kleio* Rug Pull Detection Flow

JS GitHub Actions: Whenever *Kleio* encounters a GitHub Action in the `uses` section of a workflow file, it will locally clone the Action repository and extract its whole release history. *Kleio* will then check if the Action is a JS Action or not. This check is done by seeing if the `package.json` and `package-lock.json` (or `yarn.lock`) files are present in the root of the repository. If that Action is a JS Action, then the tool will extract all the direct and indirect dependencies of that Action (i.e., the node packages). Finally, a vulnerability check is done on the version of all the direct (GitHub Actions), indirect (direct dependencies of JS Actions), and transitive (indirect dependencies of JS Actions) dependencies used by the workflow. For this check, the GitHub Advisory [14] and the OSV [15] databases are queried.

V. METHODOLOGY

After collecting all the artifacts commit histories and their vulnerabilities in the Neo4J database ("Extractor" section of Figure 4), we begin the rug pull detection process. We go through the history of each GitHub workflow and check if any JS Action is being used ("Detector" section of Figure 4). For each JS Action found, we detect rug pulls by checking if the definition described in Section III is applicable. Concretely, for each workflow commit we (i) resolve the dependencies of the current and previous commits, and (ii) we query both the GitHub Advisory and OSV databases for every package-version pair in the list of dependencies. If a new vulnerability is introduced in the current version, we record the rug pull event together with additional metadata (i.e., the location of the rug pull and the vulnerability's Common Vulnerabilities and Exposures (CVE), Common Vulnerability Scoring System (CVSS) scores, and Common Weakness Enumerations (CWE)).

Finally, we check if any of the identified rug pulls have been fixed (or can be fixable) in a commit with time $t > t_r$. In addition to identifying fixes or potential fixes, we also check which Actor (described in Figure 1) is responsible. These checks are done by seeing if the definition of a fix or possible fix (described in Section III) is applicable. As soon as a fix is found, given its type, we compute the corresponding temporal metrics (Equations 1, 2, 3, 4, and 5). The result of this whole process is then saved in a CSV file.

TABLE II
RUG PULL COUNT STATISTICS
(EXCLUDING ANYTHING WITHOUT RUG PULLS)

Aggregation	Min	Max	Med	Avg	StDev
Per RP repository	1	293	5	11.34	22.25
Per RP workflow	1	57	3	5.1	7.14
Per RP commit	1	4	1	1.23	0.46
Per RP Actions	1	2 325	4	66.69	313.36
Per RP Action Version	1	1 268	3	37.79	141.64

TABLE III
DEPENDENCY COUNTS CONSIDERING UNIQUE PACKAGE NAMES (D) OR PACKAGE NAMES AND VERSIONS (D@V)

	All		Direct		Indirect	
	JS Dep.	D@V	D	D@V	D	D@V
All	4 112	20 595	318	1 071	3 818	18 261
RP	82	390	16	49	76	354

VI. RESULTS

*RQ1: **Rug Pulls Frequency***

We detected 6 802 distinct rug pulls, present in 600 repositories (58% of our dataset), 1 135 workflows (32%), 5 527 commits (9%), 102 Actions (32%), and 180 Action versions (9%). For these rug pulled (RP) repositories, workflows, commits, Actions, and Action versions Table II shows the statistics of the number of rug pulls detected in each. Table III shows the difference between the number of unique JS Dependencies and the number of unique dependencies appearing in rug pulls. More in detail, only 2 rug pulled Actions (out of 18 820 total uses of complete tags) have a complete version tag[1], 24 (out of 908) have a branch or tag, and the remaining 6 776 (out of 112 146) have a major version. This indicates that actions using complete tags are less likely to suffer from rug pulls.

Figure 5 shows both the distribution over time of the commits and of the detected rug pulls. The marginal box plot shows that most of the rug pulls have been detected in commits spanning from October 2022 and September 2024, which roughly corresponds to the period of time in which we collected the most commits. Due to the delay in which vulnerabilities are discovered and published, the number of rug pulls for the second part of 2025 may be underestimated.

*RQ2: **Vulnerable Dependencies — Frequency and Severity***

To take a closer look at the workflows' dependencies which are causing the rug pulls, let's analyze the direct dependencies of workflows: Actions. By analyzing the actions (in our case JS Actions) that caused a rug pull, we notice that the most popular ones (Table IV) are also the ones with the most rug pulls (Figures 6 and 7). Moreover, 8 out 10 of these Actions have been developed and are maintained by GitHub. To measure the action's total usage frequency, we count all occurrences

[1] `actions/cache@v3.0.11`, and `reactivecircus/android-emulator-runner@v2.30.1`

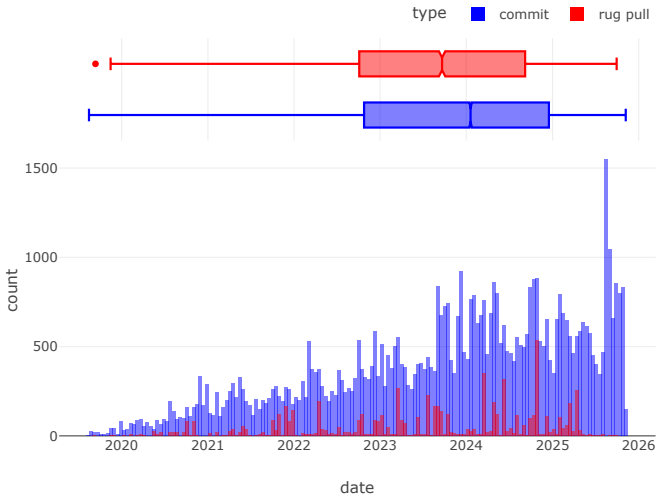

Fig. 5. Commits and Rug Pulls Distribution over Time

TABLE IV
MOST FREQUENT ACTIONS AND VERSIONS AFFECTED BY RUG PULLS COMPARED WITH TOTAL ACTIONS AND VERSIONS ACROSS ALL WORKFLOWS

Action	Usage			Versions		
	# RP	% RP	total	# RP	% RP	total
actions/checkout	2 325	4.3	54 018	36	85.7	42
actions/setup-java	1 975	5.2	37 696	40	85.1	47
actions/upload-artifact	717	5.2	17 108	29	74.3	39
actions/cache	705	3.3	20 845	45	78.9	57
actions/setup-python	111	6.4	1 715	27	64.2	42
actions/download-artifact	108	2.6	4 179	16	57.1	28
actions/setup-node	107	5.6	1 911	23	53.5	43
graalvm/setup-graalvm	52	6.5	799	8	42.1	19
actions/github-script	51	2.9	1 744	13	48.1	27
docker/login-action	46	2.3	2 021	12	70.6	17

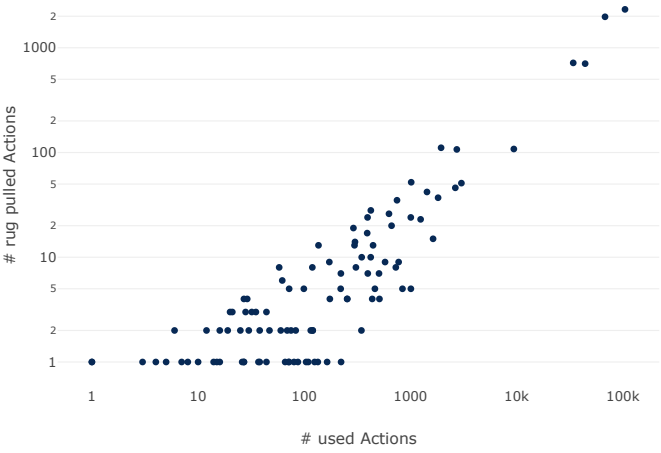

Fig. 6. Number of Action Usages in Workflows vs. How Many of them have at least One Rug Pull (Log-Log axis)

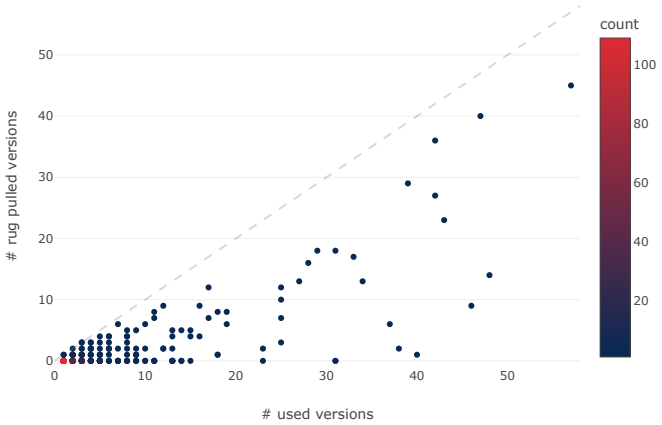

Fig. 7. How Many Action Versions have at least One Rug Pull?

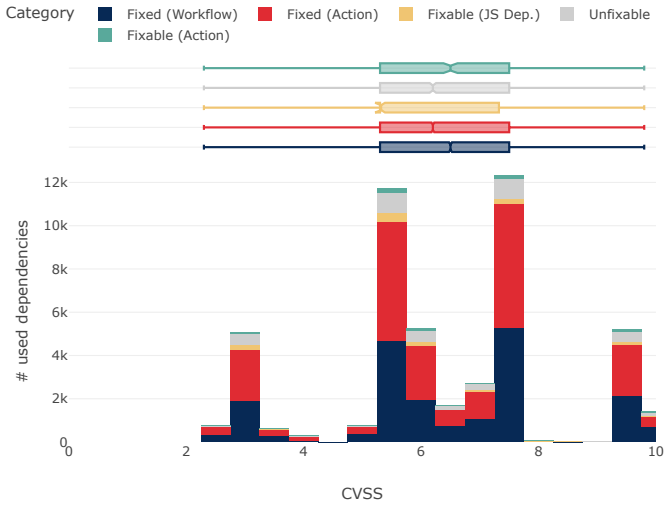

Fig. 8. Distribution of Dependency Vulnerability Severity (CVSS) classified according to the Rug Pull category

TABLE V
CVSS STATISTICS BY RUG PULL CATEGORY

Category	Min	Max	Med	Avg	StDev
Fixed (Workflow)	2.3	9.8	6.5	6.44	1.87
Fixed (Action)	2.3	9.8	6.2	6.33	1.85
Fixable (JS Dep.)	2.3	9.8	5.3	5.81	1.89
Fixable (Action)	2.3	9.8	6.5	6.66	2.07
Unfixable	2.3	9.8	6.2	6.39	1.90
All Rug Pulls	2.3	9.8	6.2	6.37	1.87

of that action being used by a workflow in our dataset. In the case of the rug pulled (RP) usages, we count how many times at least one version of that action caused a rug pull. The relationship between action usage frequency and the number of rug pulls occurring for each action is plotted in Figure 6 for all 102 Actions involved in at least one rug pull. We can see that the most rug pulled Actions are also the most used ones. For example, the most used Action is `actions/checkout` with 54 018 uses. This Action is also the most rug pulled,

TABLE VI
CVES FREQUENCIES AND PROPORTIONS GROUPED BY DIRECT AND INDIRECT USES IN GITHUB ACTIONS

CVE	CVSS	JS Dependency + Version	Direct		Indirect		Total	
			#	%	#	%	#	%
CVE-2024-4067	5.3	micromatch@v4.0.4	1	0.06	2 777	5.9	2 778	5.7
CVE-2022-25883	7.5	semver@v6.3.0	624	40.0	1 330	2.8	1 954	4.0
CVE-2025-47279	3.1	undici@v5.28.4	3	0.2	1 591	3.4	1 594	3.3
CVE-2025-22150	6.8	undici@v5.28.4	2	0.1	1 390	3.0	1 392	2.9
CVE-2025-25289	5.3	@octokit/request-error@v2.1.0	7	0.4	1 183	2.5	1 190	2.4
CVE-2025-25288	5.3	@octokit/plugin-paginate-rest@v2.21.3	29	1.8	1 145	2.4	1 174	2.4
CVE-2021-44906	9.8	minimist@v1.2.5	2	0.1	666	1.4	668	1.4
CVE-2023-0842	5.3	xml2js@v0.4.23	1	0.06	591	1.3	592	1.2
CVE-2025-54798	2.5	tmp@v0.0.33	9	0.6	485	1.0	494	1.0
CVE-2024-30261	2.6	undici@v5.28.5	1	0.06	170	0.4	171	0.4

TABLE VII
NUMBER AND PERCENTAGE OF RUG PULLS THAT CONTAIN AT LEAST ONE VULNERABLE JS DEPENDENCY OF THE GIVEN SEVERITY

CVSS	low [1, 4)		medium [4, 7)		high [7, 9)		critical [9, 10]	
Category	#	%	#	%	#	%	#	%
Fixed (Workflow)	354	5.2	1 099	16.2	836	12.3	374	5.5
Fixed (Action)	544	8.0	1 558	23.0	1 073	15.8	479	7.0
Fixable (JS Dep.)	125	1.8	257	3.8	136	2.0	69	1.0
Fixable (Action)	56	0.8	144	2.1	78	1.1	71	1.0
Unfixable	202	3.0	518	7.6	344	5.1	198	2.9
Total	1 281	18.8	3 576	52.6	2 467	36.3	1 191	17.5

with 2 325 rug pulls. In Fig. 7, we can also see on the horizontal axis (y=0) that there are some actions with many versions, none of which caused rug pulls. For example, the `step-security/harden-runner` action caused no rug pulls despite being used 207 times with 23 different versions.

Common Vulnerability and Exposures (CVEs): For each rug pulled Action, we counted the used JS Dependencies and their CVEs. In our dataset, of the 125 distinct CVEs, the most recurring is CVE-2025-27789 of the `@babel` family of dependencies, which occurred 4 768 times (10% of all dependencies). CVE-2025-27789 has a CVSS (Common Vulnerability Scoring System [16]) of 6.2 making it a medium severity vulnerability. Each rug pull category shows the same CVSS range and similar statistics (Table V). However, the histogram of the distribution of the CVSS scores for all vulnerable dependencies involved in rug pulls shows four peaks (Fig. 8). In case of direct dependencies, `@actions/core` (the JS dependencies developed by GitHub to create GitHub Actions), presents 2 different CVEs (CVE-2022-35954 and CVE-2020-15228) of low and medium severity. These two CVEs – found 843 times (747 and 96 times respectively) – together are the most present CVEs in the dataset. Regarding indirect dependencies, CVE-2025-5889 of the `brace-expansion` dependency (appearing 3 475 times) is the most present.

Only 17 JS dependencies appear in different actions both as direct and indirect. Table VI shows a list of the top 10. Regarding the percentages, the direct one is computed on a total of 1 579 (direct dependencies appearing in rug pulls), the indirect on a total of 47 216 (indirect dependencies appearing in rug pulls), and the total on 48 810 (total dependencies

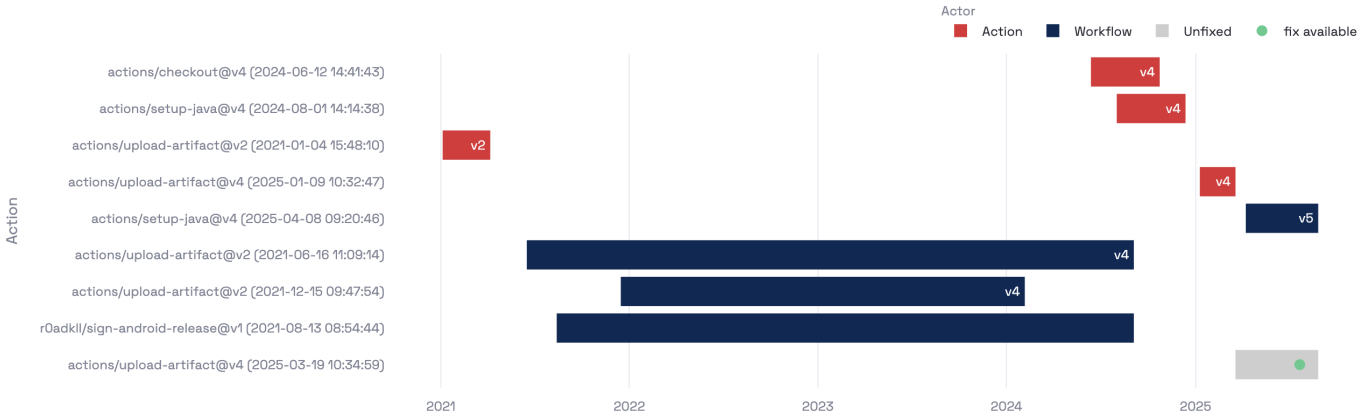

Fig. 9. Gantt Chart Showing Rug Pulls timeline in one Workflow (`wikimedia/apps-android-wikipedia/android_branch.yml`)

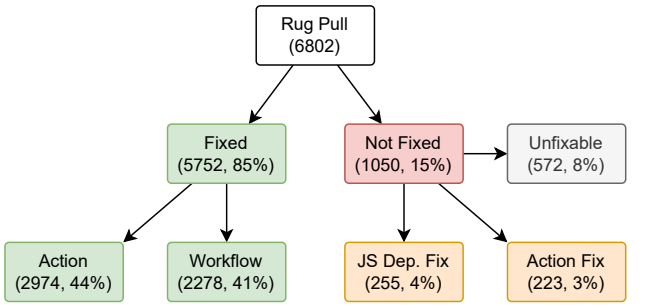

Fig. 10. Frequencies and Roles of Actors in Fixes and Potential Fixes

appearing in rug pulls). The remaining 15 have been used as direct dev dependencies. From Table VI, we can see that half are of medium severity (used 7 126 times), and one, is of critical severity (used in total 668 times).

Vulnerable dependencies have at least 1 and up to 7 different CVEs, like in case of `vm2`, which occurred 70 times in our dataset (summing up all CVEs). Moreover, there are 20 different dependencies which appear with more than 1 CVE. By looking at the frequency of CVSS in the dataset in Fig. 8 (and grouped by rug pull category), we can see how most of the vulnerabilities are fixed by Action maintainers performing reverse rug pulls. In addition, Table VII shows which severities are the most frequent in rug pulls. In our dataset, the medium severity is the most popular with 3 576 (52.6%) of rug pulls having at least one JS dependency with a severity of this type.

RQ3: ***Rug Pull Roles***

We represent all the rug pulls happening in the history of an example workflow with a Gantt chart (Fig. 9). The different commit version of actions causing a rug pull are on the y axis, time is on the x axis. Each horizontal bar in the chart indicates both when the rug pull appeared (t_r) and when it was fixed (if it was fixed). The different colors represent the actor fixing the rug pull. In case of a fix, there can be a version number at the end of the bar. In the case the version is present and it is a fix by the workflow maintainer, this means that the rug pull was fixed by having switched to that version. However, there are also cases (such as the `r0adkll/sign_android_release` action), in which there is no version shown. In such cases, the action was deleted by the workflow maintainer. Finally, there is the case where the rug pull was not fixed. A fix (at either $t_{a'}$ or t_j) can also be made available either by the Action or JS dependency maintainer (as for the last action `actions/upload-artifact`).

In Fig. 10, we classify the rug pulls depending on their final state (fixed, not fixed but potentially fixable or unfixable) and show the actors involved. For each rug pull category we indicate the absolute frequencies, and percentages. A rug pull (after t_r), can be either fixed or not fixed. In 85% of the cases, rug pulls were fixed. This fix event can be performed either by the workflow maintainer (by changing the action version to a fixed one), or by the action maintainer (by performing a reverse rug pull). In our dataset, 44% of the fixed workflows were caused by action maintainers. Not fixed rug pulls can have a possible fix available. In 4% of the 1050 not fixed rug pulls, a potential fix was published by the JS dependency maintainer, but the action maintainer did not implement the fix, which happened only in 3% of the rug pulls. Finally, 572 rug pulls (54% of the 1050 unfixed ones) remain unfixable. This means that the JS dependency maintainer has yet to provide a fix for its vulnerable dependency.

The fixable rug pulls could result from a lack of coordination between the actors involved in the rug pull. "JS Dep. Fix" means that 4% of the rug pulls had their dependencies fixed by the JS dependency maintainers. However, the action maintainers never switched to import the fixed JS dependency version. Thus, the workflow maintainers were unable to benefit from the potential fix. In "Action Fix", the action maintainers published a version (different from the one used by the workflow) with fixed dependencies. However, the workflow maintainers never noticed this fixed version and did not update the workflows, leaving their pipeline vulnerable due to the rug pull. Finally, in the "Unfixable" case, the JS dependency maintainers never provided a fix for the vulnerable dependency version, making it impossible for neither the action or the workflow maintainers to implement a fix.

*RQ4: **Temporal Metrics***

Answering the last research question requires to compute the temporal metrics for each rug pull event according to the definitions introduced in Section III: TTX (Equations 1 and 2), TTPF (Eqs. 3 and 4), and OVA (Eq. 5). In Table VIII we present the statistics for all metrics, both aggregated (TTX, TTPF, and OVA) and singularly (TTX_w, TTX_a, $TTPF_j$, $TTPF_a$, OVA_j, OVA_a, and OVA_u). In addition, we have also plotted all metrics on histograms (Figures 11, 12, and 13) to better understand and compare their distributions.

From the histogram of Figure 11, 50% of the values of TTX_a are between 60 and 216 days. This is not the case for TTX_w, where 50% of the values are between 145 and 524 days. By looking at these two distributions, we can see how the action maintainers tend to cause reverse rug pulls more quickly than their workflow counterparts upgrade the action version. This is somewhat expected since action maintainers can modify the commit under a version tag causing a reverse rug pull. In the case of workflow maintainers, since they have to actively change to the fixed version, TTX values are higher. Moreover, to the best of our knowledge, there are no tools that let maintainers know whether the action being used has an update that fixes the vulnerabilities introduced by a rug pull, making the job harder for workflow maintainers.

For unfixed rug pulls, we measure TTPF (Fig. 12), and OVA (Fig. 13). We can see that a potential dependency fix ($TTPF_j$) usually comes after 110 to 364 days from the rug pull happened. In the case of a fixed version from the action maintainer, this usually comes after 113 to 454 days. This finding underlines the delay there is in providing a fixed version to the dependents of the dependencies. When we measure the Open Vulnerability Age of rug pulls (Figure 13), we look at how long a rug pull remains unfixed for. From Table VIII, we can see that on average the rug pulls remain unfixed for 371 days, with a maximum of 1 817 days when a fix was provided by the JS dependencies maintainers.

In Figure 14, we display the distribution of the ages (in days) of rug pulls over the years. By age, we mean the time the rug pulled remained opened before it was fixed (i.e., TTX_w and TTX_a) or before the workflow's last commit (OVA_j, OVA_a, and OVA_u). From this plot, we can see, for example, that rug pulls introduced in 2019 are the only ones that have been all fixed (albeit after remaining opened for 5.4 years on average). Moreover, we can clearly see the event horizon (i.e. the constantly decreasing maximum values).

TABLE VIII
TEMPORAL METRICS STATISTICS

Metric (Days)	**Min**	**Max**	**Med**	**Mean**	**StDev**
TTX	0	1 960	175	266.65	264.26
Workflow	0	1 960	309	391.59	310.78
Action	0	1 006	116	149.94	129.04
TTPF	0	1 787	159	300.41	291.96
JS Dependency	8	1787	174	296.56	295.63
Action	0	1490	153	304.80	288.28
OVA	0	1 817	222	371.38	359.13
JS Dependency	53	1817	241	366.89	306.38
Action	21	1621	610	572.67	336.74
Unfixable	0	1780	140	294.90	359.73

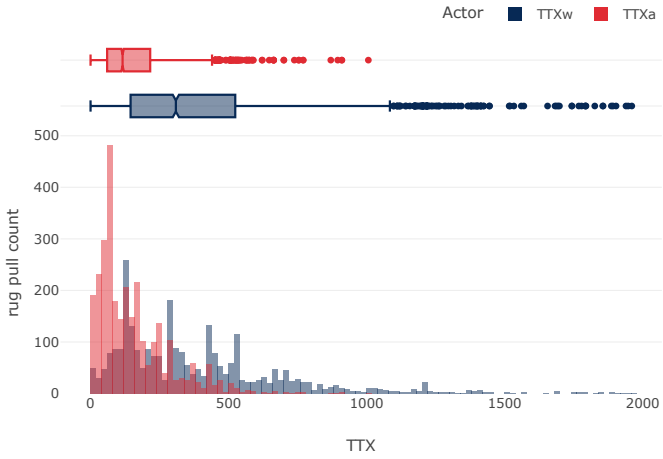

Fig. 11. Distribution of TTX_w and TTX_a Values (in Days) over all rug pulls

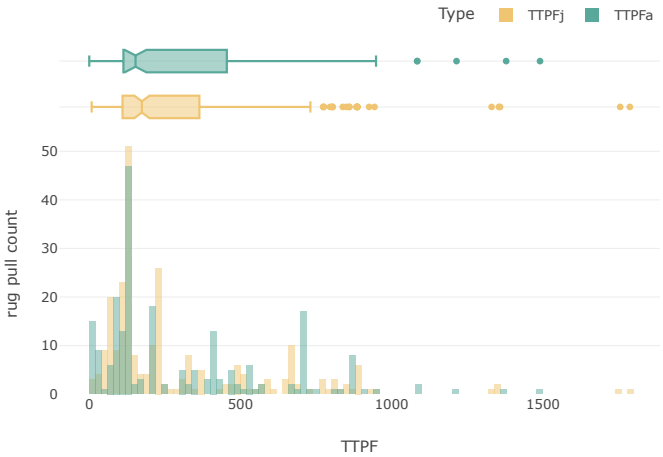

Fig. 12. Distribution of $TTPF_a$, and $TTPF_j$ Values (in Days)

VII. DISCUSSION

The results of this study provide quantitative evidence that rug pulls are a structural risk in the GitHub CI/CD ecosystem, and not just a marginal phenomenon or only a theoretical security threat. From our dataset, we were able to identify 6 802 rug pulls in more than half of the repositories affecting more than one third of the workflows. Keystone Actions like `actions/checkout`, and JS dependencies such as `brace-expansion`, account for the majority of used dependencies. This finding highlights a "single point of failure" threat, were vulnerabilities in a small number of dependencies affect the majority of the rug pulled actions. From an architectural perspective, this study calls for targeted hardening of these critical SSC components, by performing in-depth dependency checks and using hash pinning.

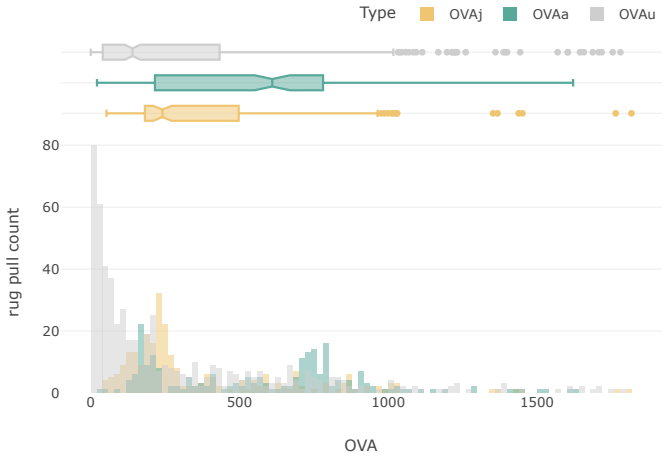

Fig. 13. Distribution of OVA_j, OVA_a, and OVA_u Values (in Days)

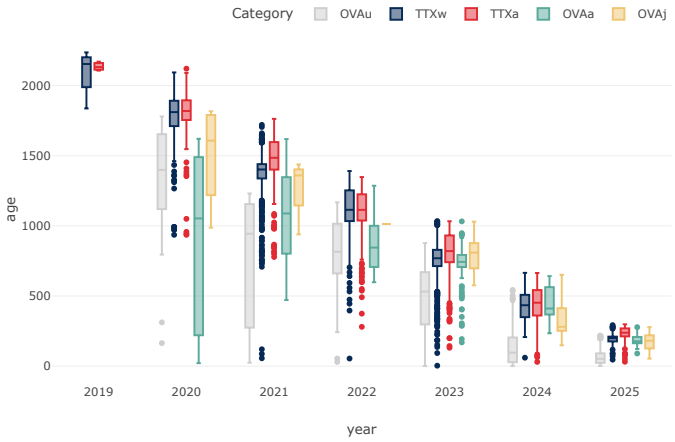

Fig. 14. Rug Pull Age (in Days) Trends over the Years

The temporal analysis highlights an asymmetry of the delays between the actors involved in rug pulls. From our findings, action maintainers tend to remediate more quickly to rug pulls than workflow maintainers. However, it's also important to notice the architectural position of action maintainers. From their viewpoint, action maintainers are in full control of the implementation of changes, without requiring per-workflow modifications (only if the workflow maintainer did not pin the action to a full-commit hash). Workflow maintainers, by contrast, must explicitly notice the release of a new (fixed) version before they can upgrade to it. This often comes at a considerable delay, or, in some cases, not at all. Currently, the architecture of the GitHub CI/CD ecosystem has a weak support for the feedback loop that goes from the JS dependency maintainer up to the workflow maintainer through the action maintainer. This is surprising since dependencies of software components are treated as first-class architectural elements of CI/CD pipelines. Dependencies of the pipelines themselves should also be treated as such.

Our findings also contribute to nuance existing discussions on version pinning and misconfigurations in GitHub workflows [7, 9, 17–19]. Our previous work [9] has classified the use of non-hash version pins (i.e. major, minor, and branch tags) as a security misconfiguration of GitHub workflows. Our analysis and results empirically confirm the associated risk by showing how these practices introduce vulnerabilities in the SSC of a CI/CD pipeline. The analysis in this paper also showed that pinning to full-commit SHAs alone is not a panacea. Workflow maintainers should still analyze the action commit to look for vulnerabilities in their dependencies. In addition, workflow maintainers could use the optional "immutable releases," where git tags cannot be removed or deleted, and release assets cannot be modified or deleted [20]. This could also suggests a direction for CI/CD platform design, where one can easily inspect the dependency graph and associated vulnerability and life cycle metadata of dependencies, even of transitive ones. Integrating SSC awareness and temporal vulnerability analytics into CI/CD tooling would help both architects and practitioners in making informed decision about reuse, upgrades, and risk acceptance.

VIII. Threats to Validity

Construct Validity: To provide a definition of a rug pull that can be detected in the context of GitHub workflows, we rely on vulnerabilities being published and correct. However, the sources we used in this study, namely OVS and the GitHub Advisory Database, could contain some inaccuracies and mismatches. There could be cases in which the vulnerability was non disclosed or retracted, in which cases could lead to report true negatives or incomplete results.

Internal Validity: The temporal metrics used in this study all rely on the detection time as one of the parameters. However, the rug pulled timestamp (t_r) is used as a proxy for when the vulnerability was introduced. This timestamps represents when the workflow maintainers became aware of the vulnerabilities as soon as the actions were rug pulled, whereas in practice there may be a lag between the event occurrence, its detection and the awareness of maintainers.

External Validity: Collecting artifacts from the top 1027 public GitHub repositories that contained at least one workflow biased the dataset towards more popular, likely better-maintained projects. Private repositories, less known repositories, and industrial CI/CD setups may exhibit different patterns than the ones we observed. Moreover, we only study the GitHub CI/CD environment, posing a threat to the generalizability of the results to other DevSecOps platforms.

IX. Related Work

The security of SSC environments has received increasing attention in recent literature, with a particular focus on the risks introduced by dependency networks within CI/CD pipelines. We contextualizes our study within the research fields of software supply chain security at the ecosystem and organizational level, security concerns specific to dependency networks, and vulnerabilities and defenses within GitHub actions and workflows.

Software Supply Chain Security

A significant body of literature focuses on broad strategies and frameworks to safeguard the SSC. A technical report [21] from the US National Institute of Standards and Technology (NIST) helps practitioners embed security in their CI/CD pipelines through automation, continuous monitoring, and governance. Other empirical studies and comprehensive surveys provide an overview of the current state of SSC security. These studies have been carried both by cybersecurity companies [22], and by researchers [23–26].

In the case of enterprises, the MITRE ATT&CK framework [27] identifies the compromise of the SSC as one of the techniques used to gain initial access in an attack. Moreover, several studies have been done to study how software supply chains are secured in practice. Angermeir et al. [2] studied how prevalent security activities are in enterprise-driven open source software, and Enck et al. [28] presented the observations from industry and government organization on the challenges of SSC security. In this paper, they also talk about the dilemma of updating a dependency version and using Software Bills of Materials (SBOM) for security.

Dependency Networks Security

Another substantial area targets the study of dependency networks cause by software reuse. Kikas et al. [29] studied the structure and evolution of the JavaScript, Ruby, and Rust dependency packages. They showed how these ecosystems are continuously growing and have become less and less dependent on singular popular packages. Another empirical study is the one carried out by Decan et al. [30] on seven different package dependency networks (i.e., Cargo, CPAN, CRAN, npm, NuGet, Packagist, and RubyGems). A final example is the study carried out by Alfadel et al. [31], where they did an empirical study focused on security vulnerabilities present in Python packages.

In the specific case of the npm ecosystem, Lauinger et al. [32] studied the presence of npm libraries in the wild, which means that they scraped npm dependencies directly from the source code of the website themselves. By doing so, they managed to see which, how, and when dependencies are actually used. Wittern et al. [33] focused more on analyzing the npm packages dependencies, as well as looking how these dependencies are used in open source repositories hosted on GitHub. Another aspect of the npm packages that has been studied is the security one. In this case we have studies focusing on the propagation and impact of security vulnerabilities in the JavaScript npm ecosystem [34, 35].

GitHub Action and Workflows Security

Many researchers have explored the characteristics and security threats present in the GitHub ecosystem, particularly dealing with workflow specifications and GitHub Actions. In the case of studies characterizing and studying the use of workflows and actions in open source repositories, we have several studies, such as the ones by Chen et al. [36] and Decan et al. [37] focusing more on workflows, and the one conducted by Decan et al. [1] specifically on GitHub Actions.

Other recent research explores the common security pitfalls in the configuration of both repositories [38] and workflows. In the case of workflows, studies have revealed some common security misconfigurations, such as incorrect version pinning of GitHub Actions [7, 9, 17–19], command injection [9, 17, 18], and emerging attack vectors like illicit cryptomining within CI environments [39]. Delicheh et al. [40] showed how the vast majority of JS GitHub Actions depend on packages with known vulnerabilities.

X. Conclusion

With this paper, we introduced and empirically studied rug pulls as a class of software supply chain attacks in GitHub workflows. We focused on JavaScript actions and their JS dependency trees. We combined the formal definition of rug pulls, fixes, and potential fixes, with a longitudinal empirical analysis of 60 287 commits from 3 543 unique workflows mined from the top starred open source repositories on GitHub. We found that a few *keystone* GitHub Actions and JS dependencies are reused thousands of times, which leads to a "single point of failure" effect. Here, exploiting the vulnerability of a single heavily used dependency affects thousand of workflows at the same time. With GitHub actions, workflow maintainers can easily pin the action to a dynamically bound version tag. This implies that, whenever an update is pushed by the action maintainer, it is instantly made available for execution in the next run of the workflow using that action. However, this comes with a significant security risk, namely the possibility of introducing rug pulls.

From the analysis of maintainers roles and temporal metrics, we found out that most of the rug pulls are eventually fixed. However, there is a non-negligible number of rug pulls that remain open for extended periods of time (with a mean open vulnerability age of roughly 1 year). In some cases, a fix is also made available by one of the dependency maintainers but never implemented by the workflow or action maintainer. From these results, we can see how coordination between maintainers across the entire stack is fundamental in the swift resolution of vulnerabilities. This implies that CI/CD pipelines designs must account for the full life cycle of reusable components, including indirect dependency monitoring, more careful pinning strategies, and mechanisms to propagate vulnerability fixes across all DevSecOps stakeholders.

Future work can build on these results in different directions. We plan to study the dependencies of the other two types of GitHub Actions, namely Docker and Composite Actions. Finally, the integration of rug pull detectors and the corresponding temporal metrics into developer- and architect-facing tools could provide a more detailed view of CI/CD supply chain risks.

Acknowledgments: This work is supported by the Swiss National Science Foundation (SNSF) funded project "Flexible Choreographies in Multi-chain Environments" (196958).

References

[1] A. Decan, T. Mens, P. R. Mazrae, and M. Golzadeh, "On the Use of GitHub Actions in Software Development Repositories," in *Proceedings of the 38th IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Oct. 2022, pp. 235–245. [Online]. Available: https://doi.org/10.1109/ICSME55016.2022.00029

[2] F. Angermeir, M. Voggenreiter, F. Moyón, and D. Mendez, "Enterprise-Driven Open Source Software: A Case Study on Security Automation," in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, May 2021, pp. 278–287. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP52600.2021.00037

[3] GitHub, "Undersdtanding GitHub Actions." [Online]. Available: https://docs.github.com/en/actions/get-started/understand-github-actions

[4] CISA, "Supply Chain Compromise of Third-Party tj-actions/changed-files (CVE-2025-30066) and reviewdog/action-setup@v1 (CVE-2025-30154)." [Online]. Available: https://www.cisa.gov/news-events/alerts/2025/03/18/supply-chain-compromise-third-party-tj-actionschanged-files-cve-2025-30066-and-reviewdogaction

[5] GitHub, "About Custom Actions." [Online]. Available: https://docs.github.com/en/actions/concepts/workflows-and-actions/custom-actions

[6] ——, "Secure Use Reference." [Online]. Available: https://docs.github.com/en/actions/reference/security/secure-use#using-third-party-actions

[7] I. Koishybayev, A. Nahapetyan, R. Zachariah, S. Muralee, B. Reaves, A. Kapravelos, and A. Machiry, "Characterizing the Security of Github CI Workflows," in *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*. USENIX Association, Aug. 2022, pp. 2747–2763. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/koishybayev

[8] A. Khatami, C. Willekens, and A. Zaidman, "Catching Smells in the Act: A GitHub Actions Workflow Investigation," in *Proceedings of the 24th IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, Oct. 2024, pp. 47–58. [Online]. Available: https://doi.org/10.1109/SCAM63643.2024.00015

[9] E. Riggio and C. Pautasso, "Pipelines Under Pressure: An Empirical Study of Security Misconfigurations of GitHub Workflows," in *Proceedings of the 26th International Conference on Product-Focused Software Process Improvement (PROFES)*. Springer, Nov. 2025, pp. 220–236. [Online]. Available: https://doi.org/10.1007/978-3-032-12089-2_14

[10] Palo Alto Networks, "Vulnerability Management Program: Building a Risk-Based Framework," 2024. [Online]. Available: https://www.paloaltonetworks.co.uk/cyberpedia/vulnerability-management-program

[11] K. Yamamoto, M. Kondo, K. Nishiura, and O. Mizuno, "Which metrics should researchers use to collect repositories: An empirical study," in *Proceedings of the 20th IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2020, pp. 458–466.

[12] E. Riggio and C. Pautasso, "Replication Package for the paper: Changing Nothing, Yet Changing Everything: Exploring Rug Pulls in GitHub Workflows," 2026. [Online]. Available: https://doi.org/10.6084/m9.figshare.30734789

[13] P. Rostami Mazrae, A. Decan, and T. Mens, "gawd: A Differencing Tool for GitHub Actions Workflows," in *Proceedings of the 21st International Conference on Mining Software Repositories (MSR)*. Association for Computing Machinery, Jul. 2024, pp. 682–686. [Online]. Available: https://doi.org/10.1145/3643991.3644873

[14] GitHub, "GitHub Advisory Database." [Online]. Available: https://github.com/advisories

[15] Google, "Open Source Vulnerabilities." [Online]. Available: https://osv.dev/

[16] NIST, "Vulnerability metrics." [Online]. Available: https://nvd.nist.gov/vuln-metrics/cvss

[17] J. Huang and B. Lin, "Revisiting Security Practices for Github Actions Workflows," in *Proceedings of the 33rd IEEE/ACM International Conference on Program Comprehension (ICPC)*. IEEE, Apr. 2025, pp. 73–77. [Online]. Available: https://doi.org/10.1109/ICPC66645.2025.00016

[18] G. Benedetti, L. Verderame, and A. Merlo, "Automatic Security Assessment of GitHub Actions Workflows," in *Proceedings of the ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED)*. Association for Computing Machinery, Aug. 2022, pp. 37–45. [Online]. Available: https://doi.org/10.1145/3560835.3564554

[19] H. O. Delicheh and T. Mens, "Mitigating Security Issues in GitHub Actions," in *Proceedings of the ACM/IEEE 4th International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS) and 2024 IEEE/ACM Second International Workshop on Software Vulnerability (SVM)*. Association for Computing Machinery, Aug. 2024, pp. 6–11. [Online]. Available: https://doi.org/10.1145/3643662.3643961

[20] GitHub, "Immutable releases." [Online]. Available: https://docs.github.com/en/code-security/concepts/supply-chain-security/immutable-releases

[21] R. Chandramouli, F. Kautz, and S. Torres-Arias, "Strategies for the Integration of Software Supply Chain Security in DevSecOps CI/CD pipelines," National Institute of Standards and Technology (U.S.), Gaithersburg, MD, Tech. Rep. NIST SP 800-204D, Feb. 2024. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-204D.pdf

[22] Snyk, "The State of Open Source Security 2024," 2024. [Online]. Available: https://snyk.io/lp/state-of-open-source-2024-dwn-typ/

[23] R. Cox, "Fifty Years of Open Source Software Supply-Chain Security," *Communications of the ACM*, vol. 68, no. 10, pp. 88–95, Sep. 2025. [Online]. Available: https://doi.org/10.1145/3762635

[24] L. Williams, G. Benedetti, S. Hamer, R. Paramitha, I. Rahman, M. Tamanna, G. Tystahl, N. Zahan, P. Morrison, Y. Acar, M. Cukier, C. Kästner, A. Kapravelos, D. Wermke, and W. Enck, "Research Directions in Software Supply Chain Security," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 5, pp. 146:1–146:38, May 2025. [Online]. Available: https://doi.org/10.1145/3714464

[25] L. Williams, "Trusting Trust: Humans in the Software Supply Chain Loop," *IEEE Security & Privacy*, vol. 20, no. 5, pp. 7–10, Sep. 2022. [Online]. Available: https://doi.org/10.1109/MSEC.2022.3173123

[26] G. Benedetti, L. Verderame, and A. Merlo, "Alice in (Software Supply) Chains: Risk Identification and Evaluation," in *Proceedings of the 15th International Conference on Quality of Information and Communications Technology (QUATIC)*, A. Vallecillo, J. Visser, and R. Pérez-Castillo, Eds. Springer International Publishing, Sep. 2022, pp. 281–295.

[27] MITRE Corporation, "MITRE ATT&CK Framework." [Online]. Available: https://attack.mitre.org/

[28] W. Enck and L. Williams, "Top Five Challenges in Software Supply Chain Security: Observations From 30 Industry and Government Organizations," *IEEE Security & Privacy*, vol. 20, no. 2, pp. 96–100, Mar. 2022. [Online]. Available: https://doi.org/10.1109/MSEC.2022.3142338

[29] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and Evolution of Package Dependency Networks," in *Proceedings*

of the 14th IEEE/ACM International Conference on Mining Software Repositories (MSR). IEEE, May 2017, pp. 102–112. [Online]. Available: https://doi.org/10.1109/MSR.2017.55

[30] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, Feb. 2019. [Online]. Available: https://doi.org/10.1007/s10664-017-9589-y

[31] M. Alfadel, D. E. Costa, and E. Shihab, "Empirical Analysis of Security Vulnerabilities in Python Packages," in *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Mar. 2021, pp. 446–457. [Online]. Available: https://doi.org/10.1109/SANER50967.2021.00048

[32] T. Lauinger, A. Chaabane, and C. B. Wilson, "Thou shalt not depend on me," *Communications of the ACM*, vol. 61, no. 6, pp. 41–47, May 2018. [Online]. Available: https://doi.org/10.1145/3190562

[33] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the JavaScript package ecosystem," in *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*. Association for Computing Machinery, May 2016, pp. 351–361. [Online]. Available: https://doi.org/10.1145/2901739.2901743

[34] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*. Association for Computing Machinery, May 2018, pp. 181–191. [Online]. Available: https://doi.org/10.1145/3196398.3196401

[35] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, and X. Peng, "Demystifying the vulnerability propagation and its evolution via dependency trees in the NPM ecosystem," in *Proceedings of the 44th International Conference on Software Engineering*. Association for Computing Machinery, Jul. 2022, pp. 672–684. [Online]. Available: https://doi.org/10.1145/3510003.3510142

[36] T. Chen, Y. Zhang, S. Chen, T. Wang, and Y. Wu, "Let's Supercharge the Workflows: An Empirical Study of GitHub Actions," in *Proceedings of the 21st IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, Dec. 2021, pp. 01–10. [Online]. Available: https://doi.org/10.1109/QRS-C55045.2021.00163

[37] A. Decan, T. Mens, and H. Onsori Delicheh, "On the outdatedness of workflows in the GitHub Actions ecosystem," *Journal of Systems and Software*, vol. 206, p. 111827, Dec. 2023. [Online]. Available: https://doi.org/10.1016/j.jss.2023.111827

[38] J. Ayala and J. Garcia, "An Empirical Study on Workflows and Security Policies in Popular GitHub Repositories," in *Proceedings of the 1st IEEE/ACM International Workshop on Software Vulnerability (SVM)*. IEEE, May 2023, pp. 6–9. [Online]. Available: https://doi.org/10.1109/SVM59160.2023.00006

[39] Z. Li, W. Liu, H. Chen, X. Wang, X. Liao, L. Xing, M. Zha, H. Jin, and D. Zou, "Robbery on DevOps: Understanding and Mitigating Illicit Cryptomining on Continuous Integration Service Platforms," in *Proceedings of the 43rd IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2022, pp. 2397–2412. [Online]. Available: https://doi.org/10.1109/SP46214.2022.9833803

[40] H. Onsori Delicheh, A. Decan, and T. Mens, "Quantifying Security Issues in Reusable JavaScript Actions in GitHub Workflows," in *Proceedings of the 21st International Conference on Mining Software Repositories (MSR)*. Association for Computing Machinery, Jul. 2024, pp. 692–703. [Online]. Available: https://doi.org/10.1145/3643991.3644899