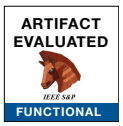

COSSETER: GitHub Actions Permission Reduction Using Demand-Driven Static Analysis

Greg Tystahl*, Jonah Ghebremichael*, Siddharth Muralee†, Sourag Cherupattamoolayil†,
Antonio Bianchi†, Aravind Machiry†, Alexandros Kapravelos*, William Enck*
*North Carolina State University, †Purdue University
Email: gttystah@ncsu.edu, jghebre@ncsu.edu, smuralee@purdue.edu, scherupa@purdue.edu,
antoniob@purdue.edu, amachiry@purdue.edu, akaprav@ncsu.edu, whenck@ncsu.edu

***Abstract*—Security vulnerabilities in GitHub Actions are increasingly leading to software supply chain attacks. In some instances, attackers have modified a project's source code by crafting a malicious issue title. To mitigate such threats, GitHub introduced a permission system that allows project maintainers to customize the privilege granted to workflows and their jobs. Unfortunately, permission policy specification is a known hard problem across nearly all domains of computing, particularly when it is introduced after an ecosystem has been established. This paper proposes COSSETER, a static analysis tool designed to determine least-privilege permission policies for jobs within GitHub Actions workflow specifications. To achieve this goal, COSSETER overcomes state explosion challenges in static analysis of JavaScript Actions that result from packing and nuances in commonly used npm dependencies. We evaluated COSSETER using a dataset of manual permission annotations of JavaScript Actions used by industry tools and found that it has a comparable precision and recall. We further evaluate COSSETER at scale, studying the permission needs of 1,842 vulnerable workflows identified by prior work and extracting permission summaries for 8,353 JavaScript Actions. We find that COSSETER's permission policy can reduce 76% of 1,274 high severity code injection vulnerabilities into medium, low, or no severity. In doing so, we demonstrate how COSSETER suggested permissions can provide a valuable defense against software supply chain attacks.**

1. Introduction

Continuous Integration (CI) systems [1] are an essential component of modern software development. There are several popular CI platforms [2], [3], [4], but GitHub Actions [5] (GitHub's CI platform) is one of the most popular due to its seamless integration with GitHub repositories and a large ecosystem of third-party modules called *Actions*. Developers use the CI platform by defining a *workflow*, which is a `YAML` file that specifies a pipeline of build steps (Listing 1 shows an example). Unfortunately, GitHub workflows are difficult to write securely, and vulnerabilities are widespread [6], [7]: Muralee et al. [8] recently discovered critical code injection vulnerabilities in 4,307 workflows and 80 Actions. Due to the tight integration with repositories, vulnerabilities in workflows have a severe impact on the security of the corresponding project source code. For example, an unprivileged attacker can open an issue with a malformed title to exploit a code injection vulnerability and modify source code files in the repository [9].

In 2021, GitHub introduced a permission mechanism to reduce the impact of workflow vulnerabilities. In February 2023, GitHub changed the default permissions to be read-only on only a few permission scopes [10]. However, this new default only applies to a repository if the associated GitHub *organization* was created in February 2023 or later.[1] Therefore, millions of repositories still have the write-all default. Furthermore, prior work in related fields (e.g., Android [11], [12], [13]) suggests that when developers are asked to specify permissions, the resulting policies are often over-privileged.

Automatically suggesting a minimal set of permissions (MINPERMS) for code execution is a classic security research problem [11], [14], [15], [16]. Approaches can be classified as *dynamic* or *static*. GitHub released a dynamic solution: *actions-permissions* [17]. However, similar to other dynamic solutions to this classic problem, it is fundamentally limited by code coverage. Furthermore, it cannot be run at scale or transparently. Many GitHub workflows rely on values in the repository's secret storage and third-party HTTP APIs. Therefore *actions-permissions* can only be run by repository owners, requiring them to add *actions-permissions* as an additional step in the workflow. Two years after its initial public release, *actions-permissions* is still in beta, perhaps suggesting limitations in its utility.

In this paper, we propose COSSETER, a static approach to the MINPERMS problem for GitHub workflows. A GitHub workflow is composed of independent units called Actions, which have well-defined inputs and outputs and can be implemented in any programming language. Most (70%) [8] of the actions are implemented through JavaScript. These Actions commonly invoke permission-protected GitHub functionality using either GitHub's HTTP REST API or its Octokit JavaScript library. JavaScript static program analysis has presented challenges in many do-

1. For personal accounts, the new read-only default applies to any repository created in February 2023 or later.

mains. Dynamic types are used liberally for both values and objects, requiring context and flow-sensitive analysis to reliably extract even a simple call graph. State-of-the-art approaches [18], [19] use abstract interpretation to generate an Object Dependency Graph (ODG) to enable more accurate control and data flow analysis. However, we found that these tools fail to analyze real-world JavaScript Actions, timing out after (in some cases) multiple days of analysis. Upon a detailed investigation, we found that statically analyzing JavaScript Actions to determine MINPERMS requires overcoming the following three challenges.

Challenge 1 - Packing: Over 50% of the JavaScript Actions used within our dataset of GitHub workflows are packed. Packing adds all npm dependencies into one file, stripping them of all metadata and replacing the require statements with an abstracted dynamic lookup mechanism. COSSETER overcomes this challenge by extending the operational semantics of state-of-the-art ODG-based static analysis tools [18], [19] and adding a pre-pass that resolves the call edges removed by packing. This pre-pass avoids significant state explosion required to resolve the call edges using abstract interpretation.

Challenge 2 - Dependency State Explosion: JavaScript Actions commonly have significant state explosion for string processing, dynamically defined objects, and other edge cases. We found that prior program slicing optimizations [19] still have timeouts due to the many functions inside of npm components that need to be resolved before the program slicing optimizations can occur. COSSETER overcomes this challenge using *demand driven analysis*. In doing so, COSSETER reduced the analysis time of the top-used `actions-checkout` Action from timing out after multiple days to completing in a matter of minutes.

Challenge 3 - Permission Extraction: There is no single, unified way to use permission protected GitHub functionality. Some approaches require a context-sensitive analysis to determine the correct permissions, such as a network call using a GitHub API route. Others require the ability to interpret `exec` calls to determine CLI commands. COSSETER handles all major cases of privileged functionality including GitHub API routes, `Octokit` methods, environment variable access, and Git CLI execution. Handling these cases allows COSSETER to extract many highly privileged permissions from Actions that would be missed from a simpler analysis. COSSETER also includes a lightweight analysis of Bash-based workflow steps to extract permissions required for raw GitHub API calls.

Results: We evaluate COSSETER in several ways. First, we compare COSSETER's ability to automatically determine MINPERMS for a given JavaScript Action to manual labeling by the company StepSecurity. While we found that neither approach is perfect, COSSETER's permission labeling is comparable to manual labeling, suggesting that it can significantly scale the process. Second, we evaluated the security impact of using COSSETER to reduce the permissions granted to GitHub workflows. Using a subset of the ARGUS dataset [8] of 1,842 workflows, we show that COSSETER can reduce the high severity of 76% by limiting permissions. Finally, we demonstrate the insufficiency of GitHub's read-only default policy. Using subsets of the broader ARGUS dataset of 1.2 million workflows and over 10,000 JavaScript Actions, we show that 50% need a permission beyond the default, thereby demonstrating the need for automated tools to help developers specify permissions.

```yaml
name: build-test-deploy
permissions:
    contents: read
on: push
jobs:
  build:
    steps:
      - name: checkout repo
        uses: actions/checkout@v3
      - name: use node.js
        uses: actions/setup-node@v3
      - run: npm install
      - run: npm run build
  deploy:
    needs: build
    permissions:
      contents: write ❌
      pages: write ✅
      id-token: write ✅
    steps:
      - name: checkout repo
        uses: actions/checkout@v3
      - run: npm install
      - run: npm run build
      - name: deploy
        id: deployment
        uses: actions/deploy-pages@v1
```

Listing 1: Example of a workflow file which builds and deploys a Next.js application on GitHub pages. Permissions denoted with ❌ are not required by this workflow; permissions with ✅ are.

Contributions: This paper makes contributions in two areas. First, COSSETER advances the state-of-the-art for static program analysis of JavaScript applications. Specifically, it (1) provides the capability to statically analyze packed JavaScript applications, and (2) further optimizes ODG construction using *demand driven analysis*. Second, COSSETER enables the static derivation of MINPERMS for GitHub workflows. Its analysis capabilities of JavaScript Actions alone has distinct value and can be used to massively expand datasets manually labeled with permissions, which are used by existing industry tools. We show GitHub's new read-only default permission policy is insufficient for 50% of workflows, and enhancing developer tooling with COSSETER suggested permissions can significantly reduce the risk of vulnerabilities in GitHub workflows. The source code for COSSETER is available at https://github.com/s3c2/cosseter [20].

2. Background

The GitHub Actions CI platform was released in 2018. It integrates directly into GitHub repositories and captures most events that users trigger while interacting with repositories. *Workflows* and *Actions* are distinct concepts in the GitHub Actions ecosystem.

TABLE 1: Permissions available in workflow specifications. We classified permissions as high, medium, or low sensitivity.

Scope	Description	High	Medium	Low
actions	Workflow and Action data such as artifacts, runs, and caching.	write	read	
contents	Stored repository code access such as merging, committing, and pushing.	write		read
checks	Check Suites functionality such adding, re-running, and changing results		write, read	
deployments	Code Deployment modifications such as statuses, adding and deleting.	write		read
discussions	Access to user discussion creation, reactions, deletion.		write	read
id-token	JWT-Token generation. Used for specific, higher authenticated routes for apps.	write, read		
issues	Repository issues creation, commenting, and deletion.		write	read
packages	Reading and changing release packages and uploads.	write		read
pages	Affect the status of GitHub Pages through uploads, health, and deletion.	write		read
pull-requests	Pull request process such as create, commenting, labeling. It does not have access to merging or changing the code of currently open pull requests. It can only create new ones.		write	read
repository-projects	Repository Project access to create, add issues, add collaborators, etc. Organization projects do not fall into this and need a PAT.		write	read
security-events	Access to code scanning alerts and scans.		write, read	
statuses	View and add commit statuses.		write	read

Workflows: Workflows are YAML files stored under the .github/workflows folder of GitHub repositories. Listing 1 shows an example workflow that is triggered on push events to the repository. As the name suggests, the workflow builds, tests, and deploys the project. Workflows are made up of several jobs (listed under the jobs key), which are independent entities to be executed in isolated environments. However, a job might depend on other jobs, and dependencies are specified using the needs key, e.g., the deploy job depends on build in Listing 1. Each job consists of a sequence of steps executed in that order. A step can be a shell command (e.g., npm install on Line 23) or can invoke Actions using the uses key, e.g., the step at Line 11 invokes actions/setup-node@v3 Action.

Actions: Actions are programs that are referenced and used within workflows to perform commonly repeated tasks. Actions are predominately written in JavaScript, but they can also be Composite (a similar syntax to a workflow) or a Docker container [21]. These Actions can be created by the developers of the workflows and stored within the workflow repository or can be third-party open-source repositories.

GitHub Token and Permissions: Workflow execution uses a capability model to grant access the repository and associated functionality. Specifically, each workflow run is passed an ephemeral GITHUB_TOKEN capability, which has associated permissions. Developers can modify these permissions within the workflow specification using the permissions keyword (as seen in Listing 1). These permissions can be specified at both the workflow level (Lines 2-3) and the job level (Lines 16-19). Job-level permissions are added to workflow permissions for the corresponding job. GitHub recommends configuring permissions at the job level, as each job could require permissions that the others do not.

Permissions are defined as <scope>:<access>, where <scope> indicates the entity and <access> indicates the type of access allowed for that entity (i.e., read or write). For instance, content:read grants read access to repository contents. Similarly, issues:write enables write access to creating or closing issues. Note that a *:write permission implicitly includes the corresponding *:read permission. There are 15 unique scopes, of which 13 are described in Table 1. GitHub recently added a new scope, attestation, which we do not consider as we did not find any uses of the scope in our analysis. metadata is also not listed, as it is included in every workflow run.

Table 1 shows our categorization of permissions into high, medium, and low sensitivity, which is based on the ease with which the permission enables modification to the repository's code or code artifacts (e.g., releases). *High* permissions enable direct modification to the code or code artifacts (e.g., contents:write). *Medium* permissions do not directly allow modifications but allows certain actions which can exploit vulnerable workflows (e.g., issues:write). *Low* permissions allow only read-only access and cannot allow modifications to the repository (e.g., pages:read).

Default Permissions: If a GitHub organization was created before February 2023, its repositories are given a GITHUB_TOKEN with write permissions for all scopes by default. For GitHub organizations created after February 2023, the default permissions were reduced to only read for the contents, metadata, and package scopes [10]. Other permissions must be explicitly declared using the permissions keyword. For repositories associated with personal accounts (i.e., not associated with an organization), the default is determined by the creation date of the repository.

3. Motivation

Code injection vulnerabilities are widespread in GitHub workflows and Actions [8], [22]. For example, Muralee et al. [8] discovered that code injection vulnerabilities in 4,307 workflows and 80 Actions can be exploited to gain complete control of the target repository by using GITHUB_TOKEN.

What We Need: An effective way to reduce the impact of vulnerabilities associated with GitHub workflows is to reduce the permissions of GITHUB_TOKEN. Specifically, *we need to provide only the required permissions (*MINPERMS*) to GITHUB_TOKEN for each job within a workflow*.

```
1  function downloadArchive(authToken, owner, repo, ref,
   ↪  commit, baseUrl) {
2    return __awaiter(this, void 0, void 0, function* () {
3      const octokit = github.getOctokit(authToken, ...);
4      const download = IS_WINDOWS
5        ? octokit.rest.repos.downloadZipballArchive
         ↪  (Contents: Read)
6        : octokit.rest.repos.downloadTarballArchive;
         ↪  (Contents: Read)
7      ...
8    });
9  }
10 ...
```

Listing 2: Simplified code from `actions/checkout` that contains a API call to GitHub endpoint using the Octokit Library (indicated by ✅)

```
1  const axios = __nccwpck_require__(6545)
2  ...
3  class Deployment {
4    ...
5    async create(idToken) {
6      const pagesDeployEndpoint = `${this.githubApi}/repos/` +
       ↪  `${this.repositoryNwo}/pages/deployment`
7      ...
8      const response = await axios.post(pagesDeployEndpoint,
       ↪  payload, { (Pages: Write)
9        headers: {
10         Accept: 'application/vnd.github.v3+json',
11         Authorization: `Bearer ${this.githubToken}`,
12         'Content-type': 'application/json'
13       }
14     }
15     ...
```

Listing 3: Simplified code from `actions/deploy-pages` that contains an HTTP API call to GitHub using network functions (indicated by ✅ where the route is created at 🚩)

The Problem: Identifying MINPERMS for a job and, consequently, a workflow is a hard problem due to the use of external Actions. Consider the workflow in Listing 1 designed to build, test, and deploy the project. At the workflow level (Lines 2-3), the `GITHUB_TOKEN` is given only `contents:read` permission, and the `build` job (Line 6) inherits this permission; intuitively, this seems reasonable as the build job needs to access the repository to build the software. The `deploy` job (Line 14) needs to deploy the latest version of the product, i.e., needs to update the `releases` tab; intuitively, this needs certain write access to the repository. Specifically, it needs `pages:write` and `id-token:write`. However, developers have unncessarily added the `contents:write` permission as indicated by ❌. Detecting such cases is nontrivial and requires an understanding of all steps within the `deploy` job.

For shell command steps, i.e., `npm ...` (on Lines 23-24), it is relatively straightforward. However, for steps that call external actions, i.e., `actions/checkout` (Line 22) and `actions/deploy-pages` (Line 27), we need to analyze their corresponding code. Few Actions do not access the repository and do not require any permissions, e.g., `actions/setup-node` [23]. However, most external Actions require specific permissions.

Listing 2 shows a simplified code snippet of the `actions/checkout` Action [24]. This action uses

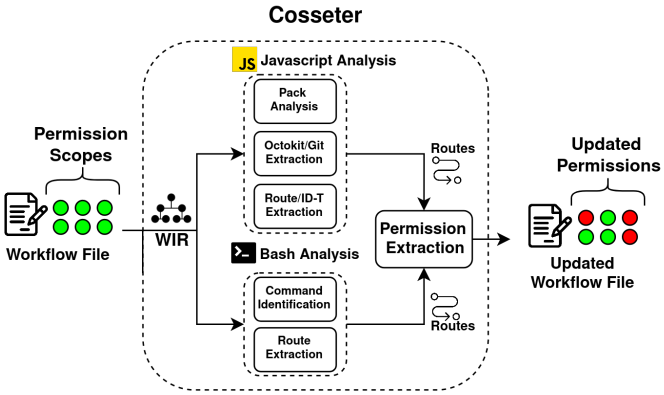

Figure 1: Overview of COSSETER

GitHub's Octokit npm library, which provides a JavaScript wrapper around nearly all GitHub API routes. The `repos.downloadZipballArchive` (on Line 5) and `repos.downloadTarballArchive` (on Line 6) calls both require the `contents:read` permission. To identify this, we need to know the permissions needed for individual functions of GitHub's Octokit npm library and identify which calls are reachable.

Not all Actions use the clearly identifiable Octokit library calls. For example, Listing 3 shows a simplified code snippet from the `actions/deploy-pages` action [25]. Instead of using Octokit, it uses the Axios network library to directly make an HTTP REST call (Lines 8-14) to a route (defined at Line 6), which requires the `pages:write` permission. Furthermore, there are accesses to environment variables that require additional permissions, e.g., to access the `ACTIONS_RUNTIME_TOKEN` environment variable at Line 6 in Listing 4, which requires `id-token:write` permission.

With the above information, we can determine that the `build` step requires only `contents:read` permissions due to its use of `actions/checkout` Action. On the other hand, the `deploy` job is over-privileged and needs `pages:write` and `id-token:write` permissions (because of `actions/deploy-pages`) and *does not need* `contents:write`.

4. COSSETER

COSSETER uses static analysis to identify a MINPERMS permission policy for each job in a GitHub workflow.

4.1. Overview

For each workflow, COSSETER begins by extracting an intermediate representation of the YAML-based workflow specification (WIR). COSSETER traverses the WIR to extract relevant information, including all jobs and steps for each job. It categorizes each step into different types, the majority being either a shell or action step. For each step, COSSETER generates a permission summary. Finally, the step-level permission summaries are combined using WIR information to produce a least-privilege permission

policy for the corresponding job. A high level overview of COSSETER is shown in Figure 1.

COSSETER's step-level permission analysis differs based on the step type. Currently, COSSETER handles two types of steps: JavaScript Actions and inline Bash. These two types of steps comprise the overwhelming majority of step types used in workflows. Prior work [8] found that 70% of Actions are developed in JavaScript. Similarly, we found that 97% of inline scripts are developed in Bash. That said, COSSETER is modular, making it easy to extend to other types of actions or shell script languages.

Handling JavaScript Actions: Step permission analysis of JavaScript Actions is the primary technical contribution of this paper. As discussed in Section 3, JavaScript Actions invoke privileged functionality in a multitude of complex ways. To identify the permissions required by a JavaScript Action, we need a precise call graph to identify these calls and the capability to identify string argument values.

As JavaScript is dynamically typed, statically extracting call graphs and string values passed to methods is non-trivial, requiring fine-grained and context-sensitive analysis. Specifically, there are two main challenges with JavaScript Actions: (1) many JavaScript Actions are packed into single files with array-based lookups to resolve function calls; and (2) significant state explosion occurs for string processing, dynamically defined objects, and other edge cases.

To overcome the challenge of packed JavaScript Actions, COSSETER performs a pre-pass to extract the key-value pairs of the lookup ID (PackID) of the array to the anonymous function that represents the packed dependency. We explain this pre-pass in more detail in Section 4.2.1. COSSETER overcomes the state explosion problem using *demand driven analysis*. Specifically, COSSETER delays performing an in-depth analysis on a statement until it is determined necessary to (a) resolve an object that would extend the control flow graph to a sink of interest or (b) resolve the value of a string passed to a network sink of interest. Our demand-driven analysis significantly reduces the number of timeouts in our experiments. We describe this analysis in Section 4.2.2.

JavaScript Actions that use network requests of HTTP route strings pose two challenges for COSSETER: (1) the HTTP route string needs to be resolved, and (2) routes resolved need to be verified as GitHub API routes to extract permissions. We use Object Dependency Graph (ODG) to capture route strings through accurate control and data flow analysis, using data dependencies to craft string values. These strings are validated using regex after extraction, which will allow future work to perform analysis on all route strings, regardless of their connection to GitHub API. This permission extraction process is described in Section 4.2.3

Handling Inline Bash Scripts: Our permission analysis of Bash scripts uses custom Semgrep [26] analysis rules to identify commonly used Bash commands. A key challenge was identifying which commands require permissions, as the current documentation lacks this information. For example, there are 10 different `git` subcommands that require `contents:read` (e.g., `git fetch-pack`) and 7 `git` subcommands that require `contents:write` (e.g., `git read-tree`, which is non-intuitive from the name). COSSETER uses the results of the Semgrep analysis to generate a permission summary for each inline Bash script. In contrast to the JavaScript Action analysis, the inline Bash script analysis directly propagates values from the workflow into the script before analysis to enhance the analysis accuracy. For example, workflows commonly define inline Bash scripts that are templated with event-driven data (e.g., pull request ID). Since COSSETER's workflow analysis is heavily based upon prior work [8], and the core technical contribution of this paper is the analysis of JavaScript Actions, we describe COSSETER's Bash analysis in Appendix A.

MINPERMS Generation: Using the summaries generated for JavaScript Action and inline Bash script steps, COSSETER generates a set of used permissions for each job within a workflow. To compute the permissions for a given job, COSSETER unions the permissions. The resulting set of permissions can either be presented to the developer as a suggestion or automatically integrated into the workflow specification. We leave this integration as an engineering effort for future work.

4.2. Analyzing JavaScript Actions

Given a JavaScript Action, our high-level idea is to construct its Object Dependency Graph (ODG) and then analyze it to identify the relevant entities (i.e., API and network calls) that require permissions and then combine these permissions.

Object Dependency Graph (ODG): ODGs are similar to Program Dependence Graphs (PDGs) [27] in that they both are graphs with directed edges of data and control dependencies. A key difference between the two is that an ODG represents JavaScript objects as nodes and creates fine-grained data dependencies between them instead of just on statements as done with a PDG. Figure 4 depicts a simplified version of an ODG with generated abstract objects. In addition to call edges, an ODG also contains argument values (e.g., via abstract interpretation), which can be used to determine URL strings for network calls. While there are many fine grained edge types between objects, Figure 4 simplifies them into a general edge for readability. ODGs were first introduced by ODGen [18] and later enhanced into ODGen-FAST [19] by adding multiple passes. However, as mentioned in Section 4.1, the prevalence of packing and deeper dependencies makes ODG construction of JavaScript Actions challenging.

4.2.1. Resolving Packed JavaScript Actions. The direct integration of GitHub Actions into the GitHub platform offers many benefits, but also contains drawbacks. Dependency resolution issues are prevalent, because GitHub is designed as a source forge and not a package manager.

NCC Packaging: Documentation on creating JavaScript Actions suggest practices to alleviate dependency issues by compiling (packing) all of the dependencies of an Action

$$\frac{\rho \Rightarrow (N,E,s,Br),(f,a.f,\rho) \Rightarrow (N_f,E_f,s_f,Br_f),(a_1,a.a_1,\rho) \Rightarrow (N_{a_1},E_{a_1},s_{a_1},Br_{a_1}),\ldots,(a_n,a.a_n,\rho) \Rightarrow (N_{a_n},E_{a_n},s_{a_n},Br_{a_n})}{(f(a_1,\ldots,a_n),a,\rho) \Rightarrow \left(\bigcup_{i=1}^{n} N_{a_i} \cup S_c \cup \bigcup_{i=1}^{n} v_{na_i}, \bigcup_{i=1}^{n} E_{a_i} \cup \{\text{AddEdge}^{s \xrightarrow{br} s}_{s \xrightarrow{br} s_c},\ \forall s_c \in S_c\} \cup E_{call} \cup E_{vo}, Sc, Br\right)},$$

$$\text{where} \begin{cases} P_{sd} := \{(\text{AddNode}^{\text{scope}}_{a'_{def}}, a'_{def}), \forall a'_{def} \in a_{def}\},\ S_c := \{p_{sd}[0], \forall p_{sd} \in P_{sd}\},\ a_{def} := \{\text{Child}^{o \to a}_{o'}, \forall o' \in \text{Child}^{a \xrightarrow{s} o}_{a.f}\},\ E_{call} := \{\text{AddEdge}^{a \xrightarrow{s} a}_{a \xrightarrow{p_{sd}[0]} p_{sd}[1]}, \forall p_{sd} \in P_{sd}\} \\ P_{vo} := \{(s_c,\ \text{AddNode}^{\text{var}}_{a.a_i}, \text{Child}^{a \xrightarrow{s_c} o}_{a.a_i}), \forall s_c \in S_c,\ \forall i \in \{1,\ldots,n\}\},\ v_{na_i} := \{p_{vo}[1], \forall p_{vo} \in P_{vo}\},\ E_{vo} := \{\text{AddEdge}^{v \xrightarrow{br} o}_{p_{vo}[1] \xrightarrow{Br} p'_{vo}[2]}, \forall p_{vo} \in P_{vo}, \forall p'_{vo}[2] \in p_{vo}[2]\} \\ \textbf{if } \{\mathbf{a'_{def}}, \forall \mathbf{a'_{def}} \in \mathbf{a_{def}}, \mathbf{a'_{def}} \in \mathbf{N_{pa}}\} \textbf{ then } \mathbf{pckScopes := S_c} \end{cases} \quad \text{(PRE CALL)}$$

$$\frac{\rho \Rightarrow (N,E,s,Br),(f(a_1,\ldots,a_n),a_{pc},\rho) \Rightarrow \rho_{pc},(B,a_B,\rho_{pc}) \Rightarrow \rho_B}{(f(a_1,\ldots,a_n),a,\rho) \Rightarrow \begin{cases} (N_{\rho_B},E_{\rho_B},s,Br) & \text{Call} \\ (N_{\rho_B} \cup \{n_{to} := \text{AddObj}^{obj}_{a}\} \cup \{n_{tv} := \text{AddNode}^{\text{var}}_{\text{"this"}}\}, E_{\rho_B} \cup E_{sv} \cup E_{vo} \cup E_{res},\ s,\ Br) & \text{New} \end{cases},$$

$$where \begin{cases} B := \{a'.B, \forall a' \in \text{Child}^{a \to a}_{a}\},\ E_{sv} := \{\text{AddEdge}^{s \to v}_{s_{ppc}} \to n_{tv}\},\ E_{vo} := \{\text{AddEdge}^{v \xrightarrow{br} o}_{n_{tv} \xrightarrow{Br} n_{to}}\},\ E_{res} := \{\text{AddEdge}^{a \xrightarrow{s} o}_{a \xrightarrow{s} n_{to}}\}, \\ \textbf{if } \mathbf{s} \in \mathbf{Child^{s \to s}_{s_p}}, \mathbf{s_p} \in \mathbf{pckScopes} \textbf{ then } \begin{array}{l} \mathbf{o_c} := \{\mathbf{o}, \mathbf{o} \in \mathbf{Child^{v \to o}_{v_c}}, \mathbf{o} \in \mathbf{Child^{a \xrightarrow{s} o}_{a}}\} \\ \mathbf{v_c} := \{\mathbf{v}, \mathbf{v} \in \mathbf{Child^{o \to v}_{o_m}}, \mathbf{v.name == call}\} \\ \mathbf{o_{pck}} := \{\mathbf{o}, \mathbf{o_m} \in \mathbf{Child^{v \to o}_{x}}, \mathbf{x} \in \mathbf{Child^{o \to v}_{o}}\} \end{array} \end{cases} \quad \text{(CALL, NEW)}$$

Figure 2: COSSETER adds packer handling to the ODG operational semantics. Bold represents COSSETER's additions.

```
8041:
(function(__unused_webpack_module, exports, __nccwpck_require__) {
  class OidcClient {
    ...
    static getIDTokenUrl() {
      const runtimeUrl = process.env['ACTIONS_ID_TOKEN_REQUEST_URL'];
      if (!runtimeUrl) {
        throw new Error('Unable to get ACTIONS_ID_TOKEN_REQUEST_URL env variable');
      }
      return runtimeUrl;
    }
    ...
    static getIDToken(audience) {
      return __awaiter(this, void 0, void 0, function* () {
        try {
          let id_token_url = OidcClient.getIDTokenUrl();
          ...
},
...
2186:
(function(__unused_webpack_module, exports, __nccwpck_require__) {
  const command_1 = __nccwpck_require__(7351);
  const file_command_1 = __nccwpck_require__(717);
  const utils_1 = __nccwpck_require__(5278);
  const os = __importStar(__nccwpck_require__(2087));
  const path = __importStar(__nccwpck_require__(5622));
  const oidc_utils_1 = __nccwpck_require__(8041);
  ...
  function getIDToken(aud) {
    return __awaiter(this, void 0, void 0, function* () {
        return yield oidc_utils_1.OidcClient.getIDToken(aud);
    });
  }
  ...
}
```

Listing 4: A simplified sample of packed JavaScript found in `actions/deploy-pages`. `8041` is the array lookup for `OidcClient`, which contains methods that require the `id-token:write` permission. `2186` is the lookup for `core`, which is referenced in multiple other listings.

together using NCC [28]. The resulting packed file becomes the new main file listed in the `action.yaml` [29].

The packing process for NCC and similar tools combines all files into one large file, replacing the original `require` statements with a dynamic abstract lookup instead. The replacement `require` function (which we call NccReq) is passed an id that is randomly generated for each dependency during packing. This id serves as an index in a map to an anonymous function that wraps the dependency. NccReq uses the id to evaluate the mapped anonymous function and generate the module object as if it were required regularly. Listing 4 shows examples of the mapping on Lines 1-18 and 20-35. It also shows an example of NccReq on Lines 22-27, with Line 27 showing a call corresponding to the mapping of `8041` on Lines 1-18.

Packing alleviates dependency resolution for JavaScript Actions, but it complicates static analysis in two ways: (1) it adds a large amount of dynamic calls, which can make it hard to statically analyze; and (2) it removes file information for the dependency, which removes metadata most analysis techniques use to reduce the analysis load. These issues are particularly problematic for ODG generation, since current generation techniques *only* support normal require statements and have no way of handling this dependency style.

Pack Dependency Analysis: To handle issues added by packing, we designed a pack dependency analysis that builds a dependency tree of packed ids to extract key value pairs of pack ids to function definition AST nodes. This is all done as a pre-pass meant to extract all pack information before generating a full ODG. Figure 3 is an example of a pack dependency tree with reductions discussed in Section 4.2.2.

We will build upon the prior work [18] to generate the ODG. Specifically, Figure 2 highlights COSSETER's changes to the operational semantics of ODG generation to extract the object that contains the mapping of pack ID to npm dependency. COSSETER begins by identifying the set of all AST definition nodes for NccReq, denoted as $N_{pa} \subset N$. Before interpreting a given AST call node, COSSETER checks to see if that call is a NccReq call by verifying that a'_{def} is an AST node found within N_{pa}. If this call is a NccReq call, then the set of scope objects (S_c) is gathered to later verify an internal call statement within NccReq. For this pass, any other call is ignored and is not interpreted. This allows COSSETER to build the dependency tree alone without having to interpret any other objects.

In NccReq, the call to the anonymous function attached to the pack id is done through the `call` prototype of the anonymous function. Therefore, to extract the map object

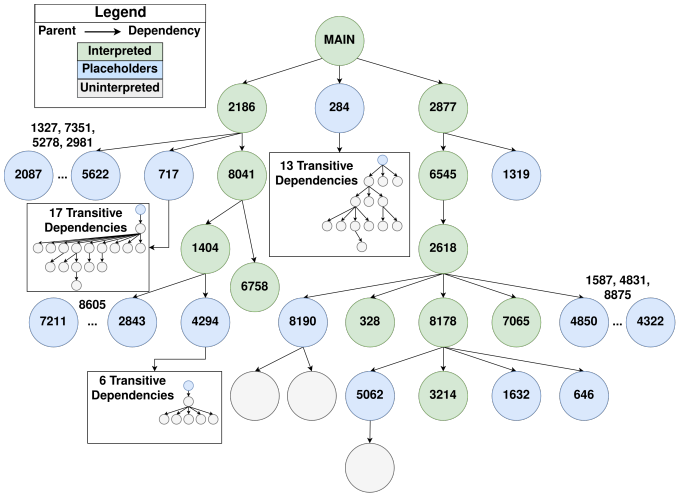

Figure 3: Demand-driven analysis drastically reduces the amount of code COSSETER needs to analyze. This figure shows a dependency tree of packed modules for `actions/deploy-pages`. Packed modules roughly correspond to npm packages.

which holds all of the pack id mappings, COSSETER must trace backwards from the call to the map object. This is done by first verifying that the current scope (s) has a direct scope edge to the NccReq scope (sp). COSSETER then traces back up the edges from the definition object of the `call` o_c. v_c represents the property name node for `call` with a parent edge to o_m which is the anonymous function definition object. With o_m, COSSETER can capture the parent property name node edge connection to x, which represents the pack id. As x is connected to o_{pck} similar to how v_c is connected to o_m, COSSETER is able to extract o_{pck} which represents the map object.

If no AST definition node is found for NccReq, then this pass is skipped, as there is no packing to handle. Once the pre-pass has finished, COSSETER extracts the internal mapping of pack id to function definition AST nodes to be used in later passes. Before moving onto the next pass, COSSETER removes all generated objects and edges begins the control flow pass from the starting AST node again.

Using the extracted map object, COSSETER creates an internal mapping of pack id to the anonymous function declaration AST node. With this mapping, COSSETER creates abstract objects to represent the module objects the anonymous functions expect as input for every pack id individually. Any exports generated during the interpretation of the anonymous function are then added as property edges to these objects. By returning the `exports` equivalent module object, COSSETER can resolve these require calls as if it were a normal require statement, including caching of already interpreted pack ids.

4.2.2. Demand Driven Analysis of Function Calls. The state-of-the-art technique [19] of building ODG is extremely inefficient for JavaScript Actions. We found several cases

```
const core = __nccwpck_require__(2186)
const { Deployment } = __nccwpck_require__(2877)
const deployment = new Deployment()
async function cancelHandler(evtOrExitCodeOrError) {
  await deployment.cancel()
  process.exit(isNaN(+evtOrExitCodeOrError) ? 1 :
  ↪  +evtOrExitCodeOrError)
}
async function main() {
  let idToken = ''
  try {
    idToken = await core.getIDToken()
  } catch (error) {
    console.log(error)
    core.setFailed(`Ensure GITHUB_TOKEN has permission
    ↪  "idToken: write".`)
    return
  }
  try {
    await deployment.create(idToken)
    await deployment.check()
  } catch (error) {
    core.setFailed(error)
  }
}
process.on('SIGINT', cancelHandler)
process.on('SIGTERM', cancelHandler)
const emitTelemetry = core.getInput('emit_telemetry')
if (emitTelemetry === 'true') {}
else {
  main()
}
```

Listing 5: The entry point for `actions/deploy-pages` whose ODG is depicted in Figure 4. The definitions of `Deployment` and `core` are in Listings 3 and 4.

where the technique does not finish even after multiple days of analysis. For example, GitHub's top-used Action (`actions-checkout`) timed out after one week. We performed a detailed investigation of possible sources of timeout and identified a range of different code idioms that caused state explosion and repeated analysis of code blocks. Furthermore, the infringing code blocks occurred in npm dependencies deep in the transitive dependency graph. In contrast, we observed that the privileged calls that COSSETER cares about were often invoked only a few levels deep in the JavaScript Action call graph. These observations led to the insight that combining demand-driven analysis of function calls with program slicing has the potential to avoid unnecessary abstract interpretation and hence avoid the state explosion leading to timeouts.

ODGen-FAST [19] generates the ODG using a two-pass approach. The first pass generates a rough control flow, and the second pass generates data flow and object dependencies. The time for the second pass depends on the complexity of the data dependencies and object accesses. Specifically, this involves transitively interpreting the abstract values for each object and its dependencies for control flow paths to sinks of interest. However, not all data flows and object dependencies for those control flow paths are needed for our permission analysis. Specifically, if an object is not involved in any call to a sensitive API or network call, then we do not need to interpret it and process its dependencies. We perform our demand-driven analysis based on this insight.

The demand-driven analysis used within COSSETER

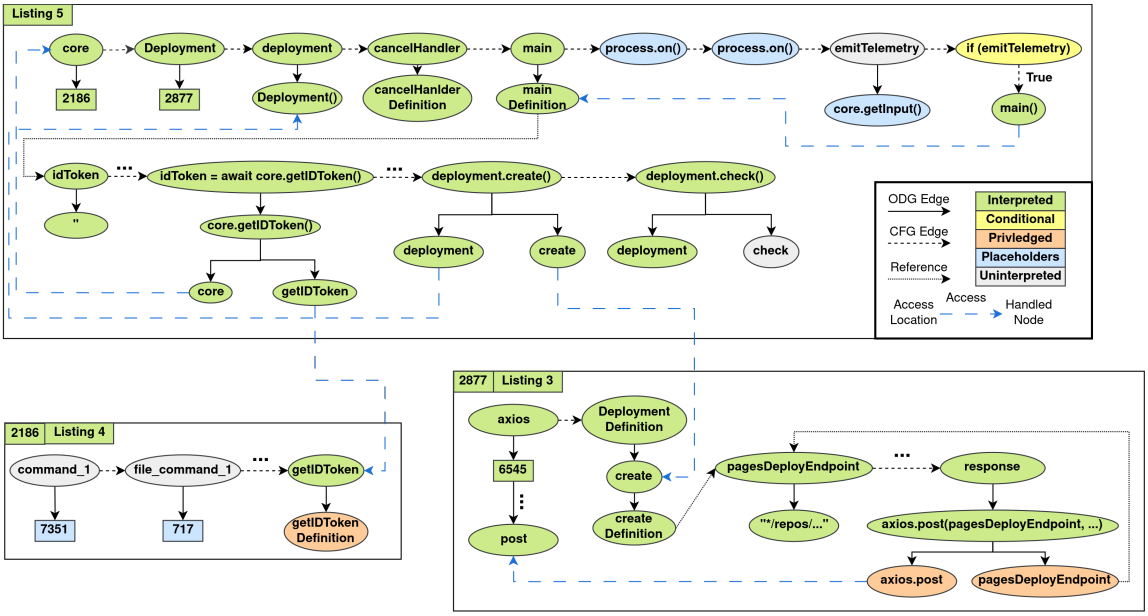

Figure 4: COSSETER uses demand driven analysis to resolve edges for the Object Dependence Graph (ODG). This figure shows a simplified ODG for the code in Listing 5. The node colors green, yellow, blue, and grey indicate the extent to which a node was analyzed in-depth when the analysis completed. COSSETER *does not* interpret blue and grey nodes.

hinges on the different uses of data as it gets interpreted or passed during the generation of the ODG. We identified two different usage types, *Access* and *Reference*, which are similar to def and use, with some key differences with respect to our analysis. *Access* refers to when an abstract object is used in the interpretation of another abstract object. In most cases, an object is accessed if it is a callee or is connected to the callee of a call statement. For instance, in a method call, the abstract object that represents the parent needs to be accessed and interpreted to generate the nodes corresponding to the method in order to be able to interpret the method call. *Reference* refers to when an abstract object gets passed without a need for interpretation, e.g., setting the value of a variable.

This *Access/Reference* model allows the analysis to delay interpreting large parts of the code until it is determined necessary for generating the inter-procedural control flow paths connected to privileged functions, or an associated value is passed as input to privileged functions. Intra-procedural control flow paths are evaluated differently, since COSSETER treats conditional statements as references. Returning an ambiguous result allows COSSETER to explore all possible *relevant* paths control flow paths extracted in the control flow pass (Pass 1).

Creating and Handling Demanded Objects: Algorithms 1 & 2 describe COSSETER's placeholder creation for dependencies. By default, any file (or packed) dependency and their export references are first created as placeholders. Module placeholders (Algorithm 1) contain all the metadata required to later interpret a dependency. Listing 5 contains two direct packed dependencies of `2186` (Listing 4) and `2877` (Listing 3).

Algorithm 1: Creation of "module" placeholder objects for demand driven analysis.

Input: ODG graph G; Call AST node of the required file a_c; Argument objects passed to the call $args$
Output: Result objects

$p_{id} \leftarrow args[0].values[0]$
if p_{id} *in* $G.packCache$ **then**
  return $G.packCache[p_{id}]$
$a_f \leftarrow$ packIdToFunc(p_{id})
$o_e, o_m, pckr \leftarrow$ packAstToArgs(a_f)
$o_f \leftarrow$ getFuncDeclObjs($astNode = a_f$)
$o_p \leftarrow G$.addBlankObject()
o_p.functionObject $\leftarrow o_f$
o_p.callAst $\leftarrow a_c$
o_p.moduleObject $\leftarrow o_m$
o_p.exportsObject $\leftarrow o_e$
o_p.packerFunction $\leftarrow pckr$
o_p.packID $\leftarrow p_{id}$
o_p.functionScope $\leftarrow G.scope$
return o_p

For `2186`, a module placeholder is created and set to the `core` variable node. Until the call is made to `getIDToken` on line 11, it remains a placeholder. The second dependency `2877` creates a module placeholder as well, but also creates an export-property placeholder for the export `Deployment` using Algorithm 2. The distinction between these placeholder types is to alleviate interpreting an entire dependency to generate an export object that has *no* data dependence for the control flow path to sinks.

Algorithm 3 describes handling when a placeholder is accessed, which begins the interpretation of the dependency

Algorithm 2: Creation of "export-property" placeholder objects for demand-driven analysis

Input: ODG graph G; Parent placeholder object o_p; Property name of the export $propName$

if $propName == *$ **then**
  Pass
else
  $o_{pn} \leftarrow$ getPropNameNode($obj = o_p, propName = propName$)
  if o_{pn} *is None* **then**
    $o_{ep} \leftarrow G$.addBlankObject()
    o_{ep}.howToHandleParent $\leftarrow o_p$
    o_{ep}.demandedProp $\leftarrow propName$
    addPropNameAndObject($obj = o_p, propObj = o_{ep}, propName = propName$)

Algorithm 3: Handler for "module" & "export-property" placeholders in demand-driven analysis.

Input: ODG graph G; Current placeholder object o_p; Abstract replacements AR
Output: Set of objects O

$dp \leftarrow o_p$.demandedProp ▷ Is none if no prop access
if *dp is not None* **then**
  $o_{old} \leftarrow o_p$
  $o_p \leftarrow o_p$.howToHandleParent
$o_f, a_c, o_m, o_e, pckr, p_{id}, s_f \leftarrow$ getPlaceholderFields(o_p)
origScope $\leftarrow$ G.current_scope
G.scope $\leftarrow s_f$ ▷ Function Scope
callFunction(o_f, a_c, [$o_e, o_m, pckr$])
$o_{ec} \leftarrow$ getPropertyObject(o_m, "exports")
for $name, exObjs$ *in getProperties(o_{ec})* **do**
  if $name == dp$ **then**
    $O \leftarrow exObjs$
end
if *dp is None* **then**
  $O \leftarrow [o_{ec}]$
else
  for $edge, typ$ *in getInEdges(o_p)* **do**
    removeEdge(o_p, $edge$)
    addEdge(o_{ec}, $edge$)
  end
  $o_p \leftarrow o_{old}$
G.scope $\leftarrow$ origScope
for $edge, typ$ *in getInEdges(o_p)* **do**
  removeEdge(o_p, $edge$)
  for $rObj$ *in* O **do**
    addEdge($rObj$, $edge$)
  end
end
$G.packCache[p_{id}] \leftarrow e_{ec}$
return O

(e.g., `2877` on line 3 of Listing 5). Since `Deployment` is an export object, O_p becomes the module object for `2877` and the metadata attached is gathered. That metadata is used to interpret `2877` as if it were being interpreted on it's original statement. Note that `6545` and any other dependencies are also created first as a placeholders and will only be interpreted if they are accessed. That is, the recursive analysis only goes as deep as needed. For this dependency's interpretation, only the definition object is created for the `Deployment` class and its method definition objects for `create` and `check`. Once interpretation is complete, $exObjs$ which represent the objects for `Deployment` are set to the output of this handle, since it was what was accessed. The edges that represented the placeholders of both `Deployment` (o_{old}) and `2877` (o_p) are removed and added to the objects $exObjs$ and o_{ec}, respectively.

As seen in Figure 4, both methods called on Lines 18 and 19 of Listing 5 are not interpreted. Only `create` is interpreted, since it is on the inter-procedural control flow path. `check` and *all* other dependencies used for that method are left uninterpreted. Dependency reduction can be shown even more for the `getIDToken` method call. Out of the six dependencies required for `2186`, only *one* dependency is needed to interpret the method to arrive at the environment access for the `id-token:write` token on Line 6 of Listing 4. Figure 3 shows the reduction of interpreted direct and transitive dependencies using these methods.

Another placeholder type is created for function calls that are on the intra-procedural control flow path, but not the inter-procedural control flow path. Previously, these functions were *fully* interpreted, as their resulting values could lead to generating objects needed to continue any control flow path. However, due to the nature of GitHub Actions, not all of the functions called along the intra-procedural path have a connection to the next statement. For instance, the `process.on` and `core.getInput` have no impact on the generated objects needed to find privileged function calls within `main`. Therefore, COSSETER skips interpreting these calls and creates placeholders that are similar to module placeholders. When the result of these calls are demanded, they are interpreted and used to continue the inter-procedural control flow path. The class constructor call for `Deployment` was created as this kind of placeholder before it was demanded to resolve `create`.

4.2.3. Resolving Permissions. There are multiple types of privileged functions that require permissions. COSSETER extracts permissions for four major types: (1) GitHub API routes passed as strings into network functions; (2) Octokit method calls; (3) Accessing environment variables that only exist when a certain permission is set; and (4) Git command line calls using exec and similar wrappers.

Route Context Resolution: The GitHub API is accessible through network requests, but not all network requests used within Actions are privileged. COSSETER requires context of the call site to be able to: (1) filter out network calls that are not related to GitHub API; and (2) capture the exact routes and or route objects being passed into network calls. As seen in the motivating Listing 3 on Lines 6 and 8, COSSETER must be able to extract the values for the variable `pagesDeployEndpoint` to determine the privileged status of the network function call `axios.post`.

COSSETER begins by finding control flow paths to any potential network related method call. These are calls such as `http.request` and are found by matching the name of the method against common network related sinks. Once

control flow paths have been found, COSSETER performs its demand driven ODG generation described above to create the fine-grained data flow to these calls. COSSETER uses the ODG to extract data dependencies of inputs passed to the calls to create the potential input strings passed. These strings are generated using a constraint solver that builds upon the dependencies generated during interpretation. The result of this constraint solver is to generate every possible combination of dependent string slices.

The extracted routes (i.e., possible strings for arguments) are passed along to COSSETER's permission extractor as shown in Figure 1. We created a mapping of all possible routes, along with the required permissions, by analyzing the official documentation – a one-time task. We match the extracted routes to our mapping to identify the required permissions. There could be cases where the extracted routes could be composed of data constraints and do not have an exact match in our mapping. We use pattern matching based on regular expressions to handle this. Examples of regex strings used can be found in Appendix C.

Octokit Method Extraction: GitHub provides the `Octokit` JavaScript library to ease the development of Actions. Octokit is an abstraction of the GitHub API. This abstraction adds to the complexity of analysis in two ways: (1) it contains many extraneous dependencies for compatibility, and (2) all of the methods with permissions are generated dynamically at runtime. While adding dependencies is no longer an issue due to our demand-driven analysis, dynamically generated methods can make it hard to statically analyze the functions they are meant to represent. During ODG generation (Pass 2), it is not possible to generate definition objects for all of the methods. Therefore, any method that does not have a definition becomes an uninterpretable method and is left out of the analysis.

While Octokit methods are dynamically generated, we found that their parent-child pairs are uniquely identifiable (eg. octokit.**repos.get**). Using these unique call sites allows COSSETER to identify if a method and parent pair has a control flow path to it in Pass 1, without having to match its function definition object by name alone. If a control flow path is found to one of these methods, the privileged function can be extracted without the need for in-depth context of the dependencies on the Octokit object or its inputs. All of the generated methods are tied to a template route, which match the permission mapping described above. COSSETER can then extract the permissions for these calls using their corresponding template route mapping without any heavy context sensitive analysis.

Privileged Environment Access: Use of the `id-token` permission (Table 1) is not done through a function call, but rather by accessing specific environment variables available during a workflow run. The token is stored as, and requested using, the environment variables `ACTIONS_ID_TOKEN_REQUEST_TOKEN` and `ACTIONS_ID_TOKEN_REQUEST_URL`.

To identify use of the `id-token` permission, COSSETER identifies all accesses to these environment variables using demand-driven analysis. Specifically, COSSETER *only* tracks environment property lookups that match the two variable names *if* there exists a control flow path to them. If a path exists, COSSETER includes the `id-token:write` permission as a required permission.

Git Command Line Usage: Actions, albeit rarely, can directly access the repository using `git` commands (e.g., `''git pull''`) through `exec` or explicit wrappers such as `GitCommandManager`. We observed that these commands are constant strings in most cases, e.g., `exec(''git pull'')`. COSSETER captures all the reachable commands by using special handlers during control flow analysis and identifies the corresponding permissions.

4.3. Implementation

We built COSSETER's functionality on top of prior work (ARGUS [8], ODGen-FAST [19]) and open source analysis tools (Semgrep [26]). We used ARGUS's WIR to extract the JavaScript Action and Bash shell step dependencies of a workflow. We implemented our Bash analysis using Semgrep, which is one of the only maintained open source static analysis tools with support for Bash scripts. We generated custom Semgrep rules to match our analysis needs.

We made significant changes to the state-of-the-art ODG generator (ODGen-FAST) for our graph generation. Aside from the implementation of our new technique, we also made changes to its parsing and handling of new JavaScript patterns found within the Actions of our dataset to reduce overall generation errors. This includes crucial builtins such as `Object.defineProperty` and TypeScript conversion functions like `__awaiter`. We also overhauled the class object handling for both Pass 1 and Pass 2, which increased our coverage dramatically. These changes increase coverage of JavaScript in newer ECMA versions. We also found and fixed major bugs dealing with function call resolution in Pass 1 across dependency files. This includes optimizations on lookup and edge creation functions to reduce redundancy. We plan to push our changes upstream upon paper acceptance.

5. Evaluation

We evaluate COSSETER via the following questions.

- **Q1** How does the precision and recall of COSSETER's automatic extraction of Action permissions compare to manual approaches?
- **Q2** What is the precision and recall of COSSETER's extraction of permissions for Bash shell steps?
- **Q3** How effective are the extracted permissions at reducing known vulnerabilities?
- **Q4** What is the landscape of permissions required by GitHub workflows?

5.1. Q1: COSSETER vs. Manual (JavaScript)

COSSETER is the first tool to automatically extract permission requirements from a given JavaScript Action. This

TABLE 2: COSSETER vs. manual approaches for JavaScript Actions. Different dataset subsets were used for each approach.

Permission	GitHub Dynamic			Step Security			COSSETER		
	C/GT	P	R	C/GT	P	R	C/GT	P	R
`actions:write`	2 / 8	**1.00**	0.25	7 / 10	**1.00**	0.70	9 / 9	0.89	**0.89**
`contents:write`	20 / 25	**1.00**	0.80	29 / 32	0.97	**0.88**	16 / 22	0.94	0.68
`deployments:write`	0 / 0	-	-	1 / 1	**1.00**	**1.00**	1 / 1	**1.00**	**1.00**
`id-token:write`	1 / 2	**1.00**	0.50	3 / 6	**1.00**	0.50	4 / 4	0.75	**0.75**
`packages:write`	0 / 1	-	0.00	2 / 2	**1.00**	**1.00**	0 / 2	-	0.00
High Permission Total	23 / 36	1.00	0.64	42 / 51	0.98	0.80	30 / 38	0.90	0.71
`actions:read`	1 / 2	**1.00**	0.50	5 / 7	**1.00**	**0.71**	2 / 4	**1.00**	0.50
`checks:write`	3 / 4	**1.00**	0.75	8 / 9	0.88	0.78	4 / 5	**1.00**	**0.80**
`checks:read`	0 / 1	-	0.00	2 / 2	**1.00**	**1.00**	2 / 2	**1.00**	**1.00**
`issues:write`	12 / 17	0.92	0.65	23 / 26	0.96	0.85	19 / 20	**1.00**	**0.95**
`pull-requests:write`	11 / 21	**1.00**	0.52	33 / 35	0.94	**0.89**	26 / 31	**1.00**	0.84
`repository-projects:write`	0 / 0	-	-	1 / 1	**1.00**	**1.00**	0 / 1	-	0.00
`security-events:write`	0 / 0	-	-	1 / 1	**1.00**	**1.00**	1 / 1	0.00	0.00
`statuses:write`	0 / 1	-	0.00	1 / 1	**1.00**	**1.00**	0 / 0	-	-
Medium Permission Total	27 / 47	0.96	0.55	74 / 83	0.95	0.84	54 / 64	0.98	0.83
`contents:read`	0 / 0	-	-	19 / 28	0.84	0.57	19 / 20	**0.89**	**0.85**
`issues:read`	2 / 1	0.50	**1.00**	2 / 3	0.50	0.33	1 / 1	**1.00**	**1.00**
`pull-requests:read`	7 / 13	0.86	0.46	12 / 14	**1.00**	**0.86**	5 / 11	**1.00**	0.45
`statuses:read`	0 / 1	-	0.00	0 / 2	-	0.00	2 / 2	**1.00**	**1.00**
Low Permission Total	8 / 14	0.75	0.43	33 / 47	0.88	0.62	26 / 33	0.92	0.73

GT = count identified in ground truth dataset; C = count identified by the tool; P = precision; R = recall

subsection compares COSSETER's precision and recall to two manual approaches: (1) the manual annotations made by a software supply chain company (Step Security) [30], and (2) executing Actions and monitoring them with GitHub's *actions-permissions* dynamic analysis tool (GitHub Dynamic) [17]. We note that using *actions-permissions* to monitor an Action requires significant manual configuration of a workflow to properly use the Action functionality.

Dataset: We created a ground truth dataset by performing a detailed manual inspection whenever the annotations of COSSETER, Step Security, or GitHub Dynamic did not agree. We started with a set 254 JavaScript Actions annotated by Step Security as of April 2024 [31]. We measured the precision and recall of each approach using different subsets. For COSSETER, we selected a subset of 188 JavaScript Actions. The remaining are either minified (48), which is not supported by COSSETER (see Section 6), or COSSETER timed out or had an error (18). For the GitHub Dynamic approach, we selected a different subset of 155 Actions. For each action, we manually created sample inputs and stopped when the Action returned a completed status. We removed actions that had errors which we could not resolve (32) or could not easily setup to run in our testing repository (19). We also removed any Actions that require a third-party API key (48). We performed a detailed manual inspection of a JavaScript Action if either (a) the annotations of the three approaches did not agree, or (b) only one annotation was found for a given action. For actions with just two annotations, we inspected them only if the two disagreed. In total, 4 graduate students with knowledge of JavaScript Actions manually inspected 140 Actions, each by at least two students.

Results: Table 2 shows the precision and recall for the three datasets. The table is organized by permission to highlight challenges in identifying different API calls. COSSETER had $\geq$ 89% precision for 12 of the 14 permissions. The low precision for the other two permissions results from only one misclassification for each permission. COSSETER's recall was similar to the manual annotation by Step Security. It had lower recall on `contents:write` due to four out-of-scope cases with respect to our analysis with an example in Appendix B. Note that `issues` and `pull_requests` are ambiguous: the same HTTP route is used for both, and the backend determines which permission is used based on the value of the identifier passed to the API. Therefore, we did not penalize COSSETER's precision for including this extra permission. The developer of the workflow can easily refine this permission based on the type of value they pass to the Action. Finally, the GitHub Dynamic approach performed the poorest of the three. A subset comparison of only actions all three tools could analyze can be found in our online appendix [32].

Takeaway: COSSETER has precision and recall comparable to manual approaches for extracting permissions from JavaScript Actions.

5.2. Q2: Bash Accuracy

In contrast to JavaScript Actions, no prior work has attempted to extract permissions from Bash steps.

TABLE 3: Vulnerable Workflow Severity Reduction

EXISTING Classification	Total Count	COSSETER Reclassification High	Medium	Low	None	Any Reduction
High	1,274	251	**266***	**511**	**246**	**964**
Medium	568	-	272	**173**	**123**	**296**

* 59 of the medium sensitive permissions were already specified by developers on the workflow. These are not included in the "Any Reduction" column.

Dataset: We started with the ARGUS [8] dataset and selected a subset of 1.4 million workflows which contained only Bash and JavaScript Actions. We randomly sampled 2,881 workflows for manual annotation by a group of 6 graduate students knowledgeable in supply chain security and GitHub workflows. Each student was tasked with searching for commands requiring permissions, e.g., `git`, `gh`, and network calls to the GitHub API. Reviewers worked with the workflow Bash step only and did not evaluate any outside scripts or action steps. At least two students annotated each workflow and differences were resolved through discussion. COSSETER analyzed 2,854 of the 2,881 sampled workflows.

Results: We measured Bash step precision and recall for the workflow as a whole instead of per permission pair. A large majority of the permissions came from `git` calls, which only corresponds `contents` permissions. 2,601 (91%) workflows did not require any permissions; COSSETER correctly extracted no permissions for 2,548. COSSETER has 53 false positives. These were mostly due to extracted `git` commands not being related to the repository containing the GitHub workflows. In total, COSSETER correctly extracted *all* permissions for 241 of 253 (95%) Bash steps.

Takeaway: COSSETER extracts all Bash step permissions in workflows with 95% recall.

5.3. Q3: Vulnerability Mitigation

ARGUS [8] found a large number of easily exploitable vulnerable workflows. It categorized all found vulnerabilities into three distinct severities: (1) High severity were vulnerabilities where attackers had full control of data (e.g. an issue title) (2) Medium severity vulnerabilities require some developer intervention to exploit (e.g. requires a specific label to run) (3) Low severity vulnerabilities give attackers limited data control (e.g. username). We now evaluate how COSSETER can reduce permissions and mitigate the impact of exploitation.

Dataset: ARGUS reported 3,687 high and 7,770 medium severity vulnerability workflows. To be conservative in COSSETER's ability to mitigate vulnerabilities, we exclude workflows that it could not handle. Specifically, we only considered workflows with JavaScript Actions and Bash steps. This consisted of 1,908 high and 3,746 medium workflows. We excluded workflows using a JavaScript action that was minified (665) or for which COSSETER timed out or had an error (237). Workflows were also excluded if the repository was missing the main file[2] (1554) or it was a mixture of minified and missing (1356). Our final dataset consisted of 1,274 high and 568 medium severity workflows.

Results: We consider a vulnerable workflow to have a reduced impact if it is assigned a permission sensitivity lower than the severity. For example, if a high severity vulnerability is given a medium or lower sensitivity permission, it has a reduced impact. To be conservative, we calculate mitigation at the *workflow* level rather than the job level.

Table 3 shows COSSETER's ability to reclassify impact severity. For the high severity dataset, developers had already specified medium sensitive permissions for 59 workflows. COSSETER suggested permissions can reduce **76%** of the vulnerable workflows (964 out of 1,274) to medium or lower permissions; 59% (757 out of 1,274) required low or lower permissions. For the medium severity dataset, the reduction was similar: 52% (296 out of 568) reduction. Low permissions on workflows gives attackers less than or equivalent abilities than any authenticated user on GitHub. Therefore, COSSETER can reduce attackers ability to affect the repository for 57% of workflows across both datasets.

Takeaway: COSSETER suggested permissions can reduce the impact of 76% of high severity vulnerable workflows.

5.4. Q4: Ecosystem-wide Permission Needs

We now study the landscape of permission needs of GitHub workflows to better understand the need for a tool such as COSSETER to help developers specify permissions. For example, we seek to understand how applicable GitHub's new default policy is for repositories created before the change, and whether or not it could be retroactively applied to older repositories.

Dataset: We again used the ARGUS [8] dataset. It is a snapshot of workflows from 2018 - 2022 (before the default change). As above, we excluded workflows with steps COSSETER cannot handle. In total, we created a dataset of 1.2 million workflows spanning 267,391 repositories. COSSETER was run on a total of 9,140 JavaScript Actions and 9,645,943 Bash shell steps. Of these Actions, COSSETER was able to extract permission summaries for 8,353 (91%) with 3,504 Actions having some kind of permission (38%). For Bash shell steps, only 17,399 (<1%) were not able to be analyzed by COSSETER and 437,854 (5%) hade some kind of permission extracted.

Results: Table 4 shows COSSETER's reported permission needs organized by permission severity. Only 52% of workflows require `Low` severity permissions. This is similar to the 50% of workflows that require only GitHub's new read-only default policy. The percentages for repositories are consistent with workflows. The other 50% requires explicit permission specification. We found that only 16% (192,498

2. The file run when an action is called is specified in the `action.yaml`. For some actions, the specified file did not exist in the repository.

TABLE 4: COSSETER Ecosystem Permission Classification

Level	Total	Classification				2023 Defaults
		High	Medium	Low	None	
Repo	267,391	109,354	18,347	134,786	4,904	133,358
Workflow	1,232,170	336,510	151,894	640,364	103,402	621,721
Job	2,152,808	675,523	178,611	1,108,679	189,995	1,055,718
Step (All)	18,035,596	1,503,515	319,936	4,143,683	12,068,462	3,994,203
JavaScript Action	8,389,653	1,296,729	289,923	3,942,628	2,860,373	3,810,586
Bash Shell	9,645,943	206,786	30,013	201,055	9,208,089	183,617

out of 1,232,170) workflows have permissions explicitly set, and only 14% (36,978 out of 267,391) of repositories set permissions for all of their workflows.

Another interesting observation is that most of the permissions are found in JavaScript Action steps (92% of all steps with permissions). This observation demonstrates that most the privileged functionality exists in third-party actions. However, contrary to JavaScript Action steps, Bash shell steps have a higher percentage of `High` classified permissions (47%) compared to (23%).

Takeaway: The 2023 default permission policy set for new repositories does not capture the needs of workflows for 50% of repositories. 92% of permission requirements come from third-party JavaScript actions.

6. Threats to Validity

Threats to Internal Validity: COSSETER extends ODGen-FAST [19]. We fixed many bugs (see Section 4.3); however COSSETER inherits all other limitations. The demand-driven approach interferes with the ability to accurately handle global variables changed within non-interpreted functions. COSSETER also misses all calls made inside of scripts called from JavaScript (e.g., via child_process, exec). Input sources from the workflows are marked within the analysis, but no attempt is made to remove calls based on any control dependence. While COSSETER captures calls made to the GitHub API, it does not consider calls made to other APIs. Therefore, an attacker can target third party data even if workflows are running with default read permissions. The integrity of build artifacts can sometimes be compromised through these APIs.

Threats to External Validity: Our datasets were constructed as subsets of workflows from prior work [8]. We further limit the dataset to workflows that only use Bash or JavaScript Action steps. We excluded workflows with composite, local, or docker Actions, or any non-Bash shell scripts. We also excluded workflows with JavaScript Actions that used minification, or for which COSSETER timed or errored out. This exclusion criteria reduced the number of workflows available to analyze, which may impact the percentage of possible vulnerable workflows that could potentially have their severity reduced.

7. Related Work

Permission Reduction: Automatically suggesting a minimal set of permissions for code execution is a classic security research problem [14], [15], [16] that continues to be relevant today [33], [34], [35], [36]. Various techniques have been employed to extract minimal permission sets, including static [37], [38] and dynamic [39], [40] based approaches. Given that permission models vary significantly across different ecosystems, researchers have developed specialized solutions for generating minimal permissions in various systems, including Linux Capabilities [41], [42], Android Applications [11], and JavaScript Applications [43]. However, these techniques are not directly applicable to GitHub Actions due to the platform's unique permission architecture and the diverse programming languages utilized in GitHub workflows.

GitHub Actions: Several studies have investigated security dimensions of GitHub Actions, including large-scale analyses of bad practices [44], code injection vulnerabilities [8], [22], [45], [46], [47], and LLM-based workflow generation and security assessment [48]. Additionally, runtime monitoring tools [49], [50], [51], [52] have been developed to dynamically observe workflow execution and detect anomalous behaviors. While existing work addresses various vulnerabilities and classifies their severity based on permissions, it has not focused on configuring workflow permissions as a primary mitigation strategy.

Javascript Analysis: JavaScript is a widely-used programming language with many applications [53], [54]. Like all programming languages, it is susceptible to vulnerabilities [55], [56], [57], [58]. To address potential issues in applications, various static [59], [60], [61] and dynamic [62], [63], [64], [65] analysis techniques are available for JavaScript. Most recently, FAST [19] proposed a scalable abstract interpretation to quickly generate ODGs of JavaScript scripts to find taint-style vulnerabilities. However, FAST times out on real-world JavaScript Actions and requires the optimizations we propose in this paper.

8. Conclusions

Vulnerabilities in GitHub Actions are a growing threat for the software supply chain. Developers can set permissions on GitHub workflows to mitigate the risk of vulner-

abilities; however, permissions require developers to understand the permission needs of third-party JavaScript Actions. We proposed the COSSETER static analysis tool that automatically determines the set of minimum permission needed by each job in a GitHub workflow. In doing so, COSSETER overcomes state explosion challenges in JavaScript Actions using demand-driven analysis and a pre-pass to handle packing. We applied COSSETER to an existing dataset of vulnerable workflows and found that permission suggestions can reduce the severity of 76% of high severity code injection vulnerabilities. Its automated analysis of JavaScript Actions can also scale efforts to annotate permission needs and immediately benefit existing software supply chain tools.

Acknowledgments

We thank the reviewers for their feedback on improving the paper. This work was supported by the US National Science Foundation (NSF) under Grants CNS-2207008, CNS-2247686, and CNS-2247688. This manuscript reflects the views of the authors and not those of NSF.

References

[1] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[2] "Travis CI - Test and Deploy Your Code with Confidence," https://travis-ci.org/.

[3] "Continuous Integration and Delivery - CircleCI," https://circleci.com/.

[4] "Set up Automated CI Systems with GitLab," https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/.

[5] "Github Actions," https://github.com/features/actions.

[6] A. Ilgayev, "How We Discovered Vulnerabilities in CI/CD Pipelines of Popular Open-Source Projects," Mar. 2022. [Online]. Available: https://cycode.com/blog/github-actions-vulnerabilities/

[7] "Vulnerable GitHub Actions Workflows Part 1: Privilege Escalation Inside Your CI/CD Pipeline." [Online]. Available: https://www.legitsecurity.com/blog/github-privilege-escalation-vulnerability

[8] S. Muralee, I. Koishybayev, A. Nahapetyan, G. Tystahl, B. Reaves, A. Bianchi, W. Enck, A. Kapravelos, and A. Machiry, "ARGUS: A Framework for Staged Static Taint Analysis of GitHub Workflows and Actions," in *Proceedings of the USENIX Security Symposium*, 2023.

[9] Tinder, "Exploiting GitHub Actions on open source projects," Jul. 2022. [Online]. Available: https://medium.com/tinder/exploiting-github-actions-on-open-source-projects-5d93936d189f

[10] GitHub, "Github actions updating the default github_token permissions to readonly," https://github.blog/changelog/2023-02-02-github-actions-updating-the-default-github_token-permissions-to-read-only/.

[11] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android Permissions Demystified," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[12] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android Permission Specification," in *Proceedings of the ACM conference on Computer and Communications Security (CCS)*, 2012.

[13] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, , and S. Weisgerber, "On demystifying the android application framework: Re-visiting android permission specification analysis," in *Proceedings of the USENIX Security Symposium*, 2016.

[14] S. Forrest, S. A. Hofmeyer, A. Somayaji, and T. A. Longstaff, "A Sense of Self for Unix Processes," in *Proceedings of the IEEE Symposium on Security and Privacy*, 1996.

[15] D. Wagner and R. Dean, "Intrusion Detection via Static Analysis," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2001.

[16] L. Koved, M. Pistoia, and A. Kershenbaum, "Access Rights Analysis for Java," in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.

[17] "GitHub token permissions Monitor and Advisor actions," https://github.com/GitHubSecurityLab/actions-permissions.

[18] S. Li, M. Kang, J. Hou, and Y. Cao, "Mining Node.js Vulnerabilities via Object Dependence Graph and Query," in *Proceedings of the USENIX Security Symposium*, Aug. 2022.

[19] M. Kang, Y. Xu, S. Li, R. Gjomemo, J. Hou, V. Venkatakrishnan, and Y. Cao, "Scaling JavaScript Abstract Interpretation to Detect and Exploit Node.js Taint-style Vulnerability," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2023.

[20] G. Tystahl, J. Ghebremichael, A. Kapravelos, and W. Enck, "Cosseter source code archive," Oct. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17345506

[21] GitHub, "About custom actions," https://docs.github.com/en/actions/creating-actions/about-custom-actions.

[22] 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)*, 2022.

[23] "Set up your GitHub Actions workflow with a specific version of node.js." https://github.com/actions/setup-node.

[24] GitHub, "Checkout actions," https://github.com/actions/checkout.

[25] "GitHub Action to publish artifacts to GitHub Pages for deployments." https://github.com/actions/deploy-pages.

[26] "Semgrep OSS is a fast, open-source, static analysis tool for searching code, finding bugs, and enforcing code standards at editor, commit, and CI time." https://github.com/semgrep/semgrep.

[27] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, 1987.

[28] vercel, "ncc simple cli for compiling a node.js module int oa single file, together with all its dependencies, gcc-style," https://github.com/vercel/ncc.

[29] GitHub, "Create a javascript action," https://docs.github.com/en/actions/creating-actions/creating-a-javascript-action.

[30] "Step Security Homepage." https://www.stepsecurity.io/.

[31] "StepSecurity knowledge base." https://github.com/step-security/secure-repo/tree/main/knowledge-base/actions.

[32] G. Tystahl, J. Ghebremichael, S. Muralee, S. Cherupattamoolayil, A. Bianchi, A. Machiry, A. Kapravelos, and W. Enck, "Permission comparison for cosseter, step security, and github dynamic," Oct. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17341657

[33] A. P. Felt, K. Greenwood, and D. Wagner, "The Effectiveness of Application Permissions," in *Proceedings of the USENIX Conference on Web Application Development (WebApps)*, 2011.

[34] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission Re-Delegation: Attacks and Defenses," in *Proceedings of the USENIX Security Symposium*, 2011.

[35] S. T. Peddinti, I. Bilogrevic, N. Taft, M. Pelikan, U. Erlingsson, P. Anthonysamy, and G. Hogben, "Reducing Permission Requests in Mobile Apps," in *Proceedings of the Internet Measurement Conference (IMC)*, 2019.

[36] K. W. Y. Au, Y. F. Zhou, Z. Huang, P. Gill, and D. Lie, "Short paper: A Look at Smartphone Permission Models," in *Proceedings of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011.

[37] J. Tang, R. Li, H. Han, H. Zhang, and X. Gu, "Detecting permission over-claim of android applications with static and semantic analysis approach," in *Proceedings of the IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2017.

[38] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon, "Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android," *IEEE Transactions on Software Engineering*, vol. 40, no. 6, pp. 617–632, 2014.

[39] P. C. Amusuo, K. A. Robinson, T. Singla, H. Peng, A. Machiry, S. Torres-Arias, L. Simon, and J. C. Davis, "Ztd$_{JAVA}$: Mitigating software supply chain vulnerabilities via zero-trust dependencies," 2024. [Online]. Available: https://arxiv.org/abs/2310.14117

[40] P. Centonze, R. J. Flynn, and M. Pistoia, "Combining Static and Dynamic Analysis for Automatic Identification of Precise Access-Control Policies," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2007.

[41] S. E. Hallyn and A. G. Morgan, "Linux capabilities: making them work," in *Linux symposium*, vol. 8, 2008.

[42] "Linux Capabilities and Seccomp." https://docs.redhat.com/en/documentation/red_hat_enterprise_linux_atomic_host/7/html/container_security_guide/linux_capabilities_and_seccomp.

[43] N. Vasilakis, C.-A. Staicu, G. Ntousakis, K. Kallas, B. Karel, A. DeHon, and M. Pradel, "Mir: Automated Quantifiable Privilege Reduction Against Dynamic Library Compromise in JavaScript," *arXiv preprint arXiv:2011.00253*, 2020.

[44] 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 USENIX Security Symposium*, Aug. 2022.

[45] "CodeQL Queries From GitHub," https://github.com/github/codeql/blob/main/javascript/ql/src/Security/CWE-094/ExpressionInjection.ql.

[46] "Raven - CI/CD Security Analyzer," https://github.com/CycodeLabs/raven.

[47] "poutine - security scanner," https://github.com/boostsecurityio/poutine.

[48] X. Zhang, S. Muralee, S. Cherupattamoolayil, and A. Machiry, "On the Effectiveness of Large Language Models for GitHub Workflows," in *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*, 2024.

[49] "Harden-Runner by StepSecuity," https://github.com/step-security/harden-runner.

[50] "Bolt Action," https://github.com/koalalab-inc/bolt.

[51] P. Kumar and V. K. Madisetti, "Sher: A Secure Broker for DevSecOps and CI/CD Workflows," *Journal of Software Engineering and Applications*, vol. 17, no. 5, pp. 321–339, 2024.

[52] "cimon action," https://github.com/CycodeLabs/cimon-action.

[53] D. Mitropoulos, P. Louridas, V. Salis, and D. Spinellis, "Time Present and Time Past: Analyzing the Evolution of JavaScript Code in the Wild," in *Proceedings of the IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2019.

[54] S. Qiu, R. G. Kula, and K. Inoue, "Understanding Popularity Growth of Packages in JavaScript Package Ecosystem," in *Proceedings of the IEEE International Conference on Big Data, Cloud Computing, Data Science & Engineering (BCD)*, 2018.

[55] C.-A. Staicu and M. Pradel, "Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers," in *Proceedings of the USENIX Security Symposium*, Aug. 2018.

[56] W. Song, Q. Huang, and J. Huang, "Understanding JavaScript Vulnerabilities in Large Real-World Android Applications," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 5, 2020.

[57] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, "Saving the World Wide Web from Vul-

nerable JavaScript," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2011.
[58] H. Onsori Delicheh, A. Decan, and T. Mens, "Quantifying Security Issues in Reusable JavaScript Actions in GitHub Workflows," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2024.
[59] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, "JSAI: a static analysis platform for JavaScript," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014.
[60] M. Madsen, B. Livshits, and M. Fanning, "Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries," in *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2013.
[61] M. Madsen, F. Tip, and O. Lhoták, "Static Analysis of Event-Driven Node.js JavaScript Applications," in *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.
[62] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An Analysis of the Dynamic Behavior of JavaScript Programs," *SIGPLAN Not.*, vol. 45, no. 6, jun 2010.
[63] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript," in *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2013.
[64] M. Pradel, P. Schuh, and K. Sen, "TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript," in *Proceedings of the IEEE/ACM IEEE International Conference on Software Engineering (ICSE)*, 2015.
[65] K. Sun and S. Ryu, "Analysis of JavaScript Programs: Challenges and Research Trends," *ACM Comput. Surv.*, vol. 50, no. 4, aug 2017.
[66] "Expressions and Operations." https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators.

Appendix A. Inline Bash Script Analysis

COSSETER's Bash analysis extracts each Bash shell step and writes it to an individual `.sh` file. During this process, workflow context (e.g., ${{github.event.issue.url}}) is replaced with variables ($VAR). This context is not part of the Bash script and therefore *must* be interpolated before Bash analysis. For example, interpolating ${{github.event.issue.url}} creates a GitHub API route such as `/repos/{OWNER}/{REPO}/issues/{ISSUE_NUMBER}` with the corresponding event data. COSSETER captures this behavior by using the template route strings found within GitHub API.

Once the GitHub actions variables are interpolated, COSSETER performs its analysis using custom Semgrep [26] rules. We designed three different types of rules: (1) capturing CLI commands (e.g., git, gh), (2) network calls to curl and wget with GitHub API routes passed as input, and (3) general regex which match to all GitHub API routes.

Our custom rules to extract `git` and `gh` command and flag pairs. These pairs correspond to the required permissions. No prior documentation or work has identified the permissions associated with these calls. Therefore, we manually defined a mapping based on command documentation. To extract the pairs, the custom rules use patterns to capture both the command and flags (e.g., git $COM ... $FLAG $VAL ...). The full list of the permission mappings and rules for `git` and `gh` can be found in our code repository.

To extract GitHub API routes, COSSETER looks for the use of `curl` and `wget` network CLI tools. The rule patterns look for two things: (a) the route being passed as input to the call; and (b) the network method attached to the url (e.g., GET, POST). As `wget` and `curl` default to GET, we also created a general pattern to extract the URL and add GET into the result (e.g., curl $URL ...) We similarly extract routes for `gh api ROUTE` calls.

Finally, we designed general regex rules that apply to all GitHub API routes. These rules capture any use of the routes missed by the above rules. The general regex strings are described in Appendix C.

The Semgrep rules provide metadata about the Bash scripts that COSSETER uses to create each step's permission summary. Similar to Actions, extracted routes are filtered and verified to a part of the GitHub API. COSSETER maps the routes and extracted commands to corresponding permissions and adds them to the summary. This summary is referenced during the workflow analysis using a combination of the workflow, job, and step ids.

Appendix B. Out-of-scope contents:write

We observed in some cases that JavaScript actions called external scripts. While we tried to handle the simpler cases of `git` using `exec` or similar functionality, cases such as the one showcased in Listing 6 were deemed out of scope for our analysis. In this example, the action is dynamically downloading an external dependency `electron-builder` on line 11 provided by the users defined `package.json` from line 9. Using this newly downloaded dependency, it publishes a new release to GitHub by running the constructed `cmd` on line 16. Along with this, any external script called from JavaScript actions in as similar way is also out of scope for our analysis (e.g. `.sh` files).

Appendix C. Regex For GitHub API Routes

COSSETER extracts route strings from various sources. The extracted routes do not perfectly match the route

```
const run = (cmd, cwd) => execSync(cmd, { encoding: "utf8",
↪  stdio: "inherit", cwd });
const getInput = (name, required) => {
        const value = getEnv(`INPUT_${name}`);
        if (required  !value) {
                exit(`"${name}" input variable is not
                ↪  defined`);
        }
        return value;
};
const pkgRoot = getInput("package_root", true);
log(`Installing dependencies using ${useNpm ? "NPM" :
↪  "Yarn"}...`);
run(useNpm ? "npm install" : "yarn", pkgRoot);
log(`Building${release ? " and releasing" : ""} the Electron
↪  app...`);
const cmd = useVueCli ? "vue-cli-service electron:build" :
↪  "electron-builder";
for (let i = 0; i < maxAttempts; i += 1) {
    try {
        run(
            `${useNpm ? "npx --no-install" : "yarn run"}
            ↪  ${cmd} --${platform} ${
                release ? "--publish always" : ""
            } ${args}`,
            appRoot,
        );
        break;
    } catch (err) {
        if (i < maxAttempts - 1) {
            log(`Attempt ${i + 1} failed:`);
            log(err);
        } else {
            throw err;
        }
    }
}
...
```

Listing 6: A code simplified snippit from `samuelmeuli/action-electron-builder` which dynamically installs electron-builder and uses it to publish a new GitHub release

TABLE 5: Demand Handler Types for Expressions

Expression	Example	Type
Property accessors	MP.AP	reference
Property accessors	AP.property	access
Property accessors	Any[AP]	access
Function Call	AP()	access
Function Arguement	function(AP)	access
Addition (+)	AP + const	access
Addition (+)	AP + AP	access
Addition assignment (+=)	var += AP	access
Addition assignment (+=)	AP += exp	access
Assignment (=)	var = AP	reference
await	await AP()	access
Conditional (ternary) operator	exp ? AP : Any	reference
Conditional (ternary) operator	AP ? Any : Any	reference
Decrement (–)	AP–	access
delete	delete object.AP	reference
delete	delete AP.property	access
Destructuring assignment	[a, b] = object.AP	reference
Destructuring assignment	[a, b] = AP.properties	access
Equality Operators (==, !=, >, <)	Any <= Any	reference
Increment (++)	AP++	access
Logical Operators (&&, ‖, !)	AP && Any	reference
new	new AP()	access
Object initializer	object = Any: AP	reference
Strict equality (===)	AP === Any	reference
Strict inequality (!==)	AP !== Any	reference
Subtraction (-)	AP - Any	access
Subtraction assignment (-=)	AP -= Any	access
Unary negation (-)	-AP	access
Unary plus (+)	"+AP"	access
void operator	void (AP)	access
yield	yield AP	access
while	while(AP)	reference
for	for (AP, cond, inc)	reference
for	for (init, AP, inc)	access
for	for (init, cond, AP)	access
for in	for (v in AP)	access
for of	for (v of AP)	access
if else	if (AP) else	reference
switch	switch (AP)	access

AP: Any Placeholder, MP: Module Placeholder Specifically, Any: Any Object

templates listed in the GitHub API documentation. Therefore, we generated regex strings that match each route in the GitHub API. For example, the regex string `(/?repos/([^/]+)?/?([^/]+)?/labels/[^/\s'"$]+)` corresponds to the `/repos/owner/repo/labels/name` route. In some cases, the regex for one route matches a longer version of the route (/repos/owner/repo/issues vs /repos/owner/repo/issues/issue). When this occurs, COSSETER uses the longer regex match.

Matching a route string to the template is insufficient to identify permission needs. The permissions for some routes depend on the network method (i.e., GET, POST, etc). Therefore, COSSETER also obtains the method. When the method is not given, COSSETER defaults to `GET` *if* the route contains `api.github.com`. This default increases the route extraction precision and reduces false positives not related to the GitHub API.

Appendix D. JavaScript Network Sinks

Cosseter has the ability to resolve API strings passed to network functions. These functions are discovered in two ways: (1) Direct function calls (e.g. `fetch(API)`); (2) Method function calls (e.g. `https.get(API)`). Both discovery methods are looking for functions with the following names: `paginate`, `request`, `fetch`, `curl`, `get`, `post`, `put`, `patch`, `del`, `getJson`, `postJson`, `putJson`, and `patchJson`. Direct function call discovery is mostly straight forward for both passes, but method function discovery has some nuance. While data-dependent relations cannot be accurately discovered in the control flow pass, Cosseter considers any method call which matches a name above as potential sinks except for `get`. Since `get` is used pervasively outside of network specific calls, Cosseter requires that it must be attached to an object passed in from a dependency.

Appendix E. Access / Reference for JavaScript Expressions

To perform the demand-driven analysis, COSSETER uses hooks to determine when a given ODG object handle is an *Access* or a *Reference*. We manually annotated expression types using JavaScript language documentation [66]. Table 5 shows the list of all of the expressions and the corresponding *Access / Reference* label.

Appendix F.
Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

F.1. Summary

The paper presents Cosseter, a static analyzer for GitHub actions and workflows that automatically determines the least privilege permissions required for any GitHub resources those actions and workflows access. Cosseter proposes the first practical dependency analysis for JavaScript actions. The evaluation over 1 million actions and workflows shows that Cosseter performs at least as well as manually annotated datasets.

F.2. Scientific Contributions

- Provides a New Data Set For Public Use
- Creates a New Tool to Enable Future Science
- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field
- Establishes a New Research Direction

F.3. Reasons for Acceptance

1) Cosseter represents the first practical automated static analysis of JavaScript actions for permission identification.
2) The authors plan to open-source Cosseter, supporting future research building on their work.

F.4. Noteworthy Concerns

1) The paper does not fully validate that the permissions suggested by Cosseter preserve the functionality of GitHub actions and workflows.
2) The evaluation does not answer the question of whether Cosseter, or manual annotations, actually compute least privilege permissions.
3) Table 2 does not report results for a common subset that all three evaluated methods can handle, which would help interpret the effectiveness of the approach.
4) The approach does not support invocation of external Bash scripts which are commonly used in GitHub actions.

Appendix G.
Response to the Meta-Review

Concern 3: We considered several ways to present the results presented in Table 2. We believe the current table provides the fairest comparison of the three approaches (COSSETER, Step Security, and GitHub Dynamic). Our online appendix [32] provides a full break down of the permissions extracted by each approach for the subset of 116 GitHub Actions.