More Haste, Less Speed: Cache Related Security Threats in Continuous Integration Services

Yacong Gu*†, Lingyun Ying‡†✉, Huajun Chai‡, Yingyuan Pu‡, Haixin Duan*†, Xing Gao§
*Tsinghua University, ‡QI-ANXIN Technology Research Institute,
§University of Delaware,†Tsinghua University-QI-ANXIN Group JCNS
*{guyacong, duanhx}@tsinghua.edu.cn, ‡{yinglingyun, chaihuajun, puyingyuan}@qianxin.com, §xgao@udel.edu

Abstract—**Continuous Integration (CI) platforms have widely adopted *caching* to speed up CI task executions by storing and reusing dependent packages. Unfortunately, CI cache also exposes new attack surfaces when cache objects are shared across trust boundaries. In this paper, we systematically investigate potential security threats of CI cache features in seven mainstream CI platforms (CIPs). We find that existing CIPs have isolation issues in their cache sharing and inheritance strategies, potentially raising cache poisoning and data leakage problems. By exploiting these vulnerable mechanisms, we further uncover four attack vectors enabling attackers to stealthily inject malicious code into the cache or steal sensitive data. Even worse, many CIPs provide vulnerable official cache templates that will mistakenly store and expose sensitive data in the cache by default. To understand the potential impact of our disclosed threats, we develop an analysis tool and conduct a large-scale measurement on open-source repositories. Our measurement results show that many popular repositories are potentially affected by these attacks. We also identify 78 repositories that expose their high-value secrets in cache objects and are at risk of secret leakage. We have duly reported identified vulnerabilities to corresponding stakeholders and received positive responses.**

1. Introduction

Continuous Integration (CI) is a software development practice for automated code build, integration, and testing. CI provides an efficient approach to integrating the work of different developers, greatly simplifies their daily work, and shortens the software development period. It is reported that 70% of organizations have adopted CI workflow to package and release new software versions [1].

CI's efficiency, particularly task execution efficiency, is one key consideration for developers [2]. One of the most effective ways to make CI tasks faster is *caching* [3], which has been widely used and supported by almost all mainstream CI services [4]. For example, CI cache can store dependent packages during the execution of a CI task. Then, subsequent CI tasks can reuse these package files from the cache instead of re-downloading them from the package registries (e.g., npm and PyPI).

As CI plays an important role in software development, its security is essential to ensure software security. If attackers can break the CI workflow, they can inject malicious code into the software package to launch various software supply chain attacks [5], [6], [7]. Moreover, sensitive data (e.g., *secrets*) is often used and stored in CI. Improperly configured CI with weak isolation mechanisms might leak sensitive data, causing serious security risks [8]. Furthermore, the improper usage of CI cache can also raise severe security threats. For example, if CI caches are not properly separated among different trust levels (e.g., different repositories), attackers can implant malicious code in the shared cache to attack other victim repositories. While extensive research efforts have been conducted on CI security [4], [8], [9], [10], unfortunately, little attention has been paid on cache related security threats in CI services.

In this paper, we conduct the first systematic study toward understanding security threats in CI caches. We identify two widely adopted CI caching mechanisms that can potentially lead to serious security risks. The first one is cache *sharing*: in many CI services, repositories (potentially from different users) might share the same CI cache. A cache object generated by a repository may be used by other repositories. Attackers can initiate a cache poisoning attack by generating a cache object (e.g., package) containing malicious code, which may be injected into victim software without accessing their repositories. The second vulnerable mechanism is cache *inheritance*: a forked repository might inherit its parent repository's cache without any configuration needed. Although the inheritance is one-way only (i.e., read-only), it enables a CI task triggered by a pull request from a forked repository to potentially exfiltrate sensitive data (e.g., *secrets*) from the parent repository.

We analyze the cache implementation of seven commonly used CI Platforms (CIPs), comprising three built-in CIPs (GitHub Actions, GitLab CI, and Bitbucket Pipelines) and four independent CIPs (CircleCI, TravisCI, TeamCity, and Jenkins), across three major Code Hosting Platforms (CHPs) (GitHub, GitLab, and Bitbucket). We carefully examine their mechanisms of cache authentication, storage, isolation, and inheritance. We identify four major cache-related security

✉Lingyun Ying is the corresponding author.

issues in existing CIPs: (1) Improper write-up: a low-level permission CI job (e.g., triggered by a pull request from the forked repository) can potentially poison caches used by high-level permission CI jobs. (2) Improper read-up: a low-level permission CI job can read cache objects sourced from a high-level permission CI job, causing sensitive data leakage. (3) Improper token permission: the token for cache access authentication (i.e., T_{cache}) may not be properly limited to the cache of the originated repository, leading to privilege escalation. (4) Improper cache eviction: some CIPs might not timely evict cache objects. A cache object will remain valid even after its creator has lost related privileges (e.g., a fired employee or the old owner of a transferred project). This issue potentially allows such attackers to implant malicious code (e.g., a backdoor) in the cache and affect the repository even after they have left.

From our analysis, we find that all seven CIPs have problems with their cache isolation strategies. Three CIPs (i.e., GitLab CI, CircieCI, and TravisCI) provide vulnerable official cache templates, which include misconfigured cache rules. These templates will mistakenly store sensitive data in the cache, such as Maven registry login credentials and Cargo registry API tokens. Moreover, we find that many binary tools (e.g., `docker`, `aws`), CI plugins (e.g., `docker/login-action`), and customized scripts used in CI tasks will also implicitly store secrets to cached files during task execution. Finally, we find that TravisCI and CircleCI use over-privileged T_{cache}. Attackers can steal the T_{cache} by initiating a pull request and further manipulate the victim's cache objects directly. Based on these security threats, we further propose four attacks against CI cache, enabling attackers to access unauthorized resources, steal tokens, and implant malicious code into software artifacts.

To evaluate the potential impact of our proposed attacks, we design and develop an analysis tool, *CAnalyzer*, to conduct a large-scale measurement on the three CHPs. *CAnalyzer* collects open-source repositories and their metadata (such as cache usage and permission settings) from CHPs, including data from January 2017 to January 2023. Then it utilizes predefined rules to detect potential cache vulnerabilities, such as privilege escalation and cache poisoning. *CAnalyzer* further uses variable analysis and data flow analysis to find sensitive files (i.e., files containing secrets) in the cache to detect data leakage vulnerabilities.

Our measurement results show that 78 repositories leak highly sensitive secrets in the CI cache, such as software signing certificates used to sign database management software with over 100 million downloads, and credentials used to publish popular Docker images with millions of downloads. Besides, many repositories, including some very popular repositories and repositories from large organizations (e.g., Microsoft and Google), are potentially vulnerable to other uncovered threats. We have disclosed our findings to impacted stakeholders and received positive feedback.

In summary, the major contributions of this work include:

- We present the first systematic study on cache-related security threats in CI services. We identify several risks involving cache sharing and isolation and find that existing mainstream CIPs are all susceptible to these security risks, which could result in severe consequences.
- We unveil four attack vectors allowing attackers to poison the CI cache, inject malicious code (e.g., a backdoor), and steal high-value sensitive data (e.g., *secrets*). We design and develop a fine-grained analysis tool, *CAnalyzer*, to conduct a large-scale measurement on three widely used CHPs, revealing that our proposed attacks may present significant security risks to CI users.
- We have disclosed all our findings to the impacted stakeholders and received positive feedback.

2. Background

2.1. CI Introduction

A typical CI workflow involves several independent stakeholders, including code hosting platforms (CHPs), CI platforms (CIPs), and third party services (TPSs). CHPs are widely used to manage source code in repositories. Popular CHPs generally adopt role-based access control to their repositories (e.g., owner and collaborator). CIPs are responsible for executing CI tasks of a repository on runner hosts. A CIP may either be integrated into a CHP (such as GitHub Actions) or exist independently of CHPs (such as CircleCI). Repository owners can configure CI tasks via a configuration file (e.g., `.travis.yml` in TravisCI) or a web portal (e.g., TeamCity). Two important configurable elements of a CI task include: (1) *Execution triggers*, which specify the events that will trigger a CI task. For instance, developers can specify a push event or pull request as an execution trigger. (2) *Secrets*, which can be used to access TPSs during the CI task execution. On most CIPs, the recommended way of using secrets is to define them as key-value pairs on CIP's web portal and use keys instead of values (e.g., plaintext passwords) in the CI task configuration to avoid exposing secrets [11], [12]. According to the CI task model, each CI task consists of a set of *jobs*, which further contains a sequence of *steps*. Each step includes one or more commands, such as invoking shell tools. Among them, job is the smallest unit of CI permissions [13], [14]. All steps in the same job have the same permissions, but different jobs of the same CI task may have different permissions. For example, GitHub Actions starts a separate fresh virtual machine for each job and only provides secrets to explicitly specified jobs.

2.2. CI Cache Categories

All mainstream CIPs support caching. CI cache can be used in many scenarios, including among (1) different jobs of the same CI task; (2) different CI tasks; and (3) different repositories. There are mainly two cache work modes: proactive cache and passive cache.

Proactive Cache. Proactive cache means that developers should explicitly enable the cache feature (e.g., ❶ in Listing 1) and specify the cached files by *cache rules* (i.e., paths) in the CI configuration file (e.g., ❷ and ❺ in Listing 1). At the

$$\underbrace{\text{s3://travisci**}}_{\textbf{S3 bucket name}}/\underbrace{\text{5724562189**}}_{\textbf{Repository id}}/\underbrace{\text{master}}_{\textbf{Branch}}/\underbrace{\text{cache-**}}_{\textbf{Cache key}}\text{.tgz}$$

Figure 1: An example of cache object storage path in TravisCI.

same time, developers can specify a cache key (e.g., ❸ and ❹ in Listing 1) for the corresponding cache object. When a CI task starts, based on the configuration file, CI services will check (and obtain) the existing cache object for the specific cache key. Similarly, before the CI task ends, CI services will package the specified cached files into a compressed file (i.e., cache object) and save it to the predefined location.

The cache storage can be (1) a local directory in the CI runner host or (2) a distributed location of third-party storage services (e.g., Figure 1 shows an example of TravisCI's cache storage path on AWS S3). If stored in local storage, the cache objects can only be used for subsequent CI tasks running on the same runner host. For distributed storage (e.g., S3), the cache objects can be used by any subsequent CI tasks (running on any runner hosts) with the valid access token (i.e., T_{cache}). GitHub Actions, GitLab CI, Bitbucket Pipelines, CircleCI, and TravisCI adopt proactive cache. Among them, GitLab CI supports both local and distributed storage, while the others only support distributed storage.

```
# GitHub Actions Cache Example
steps:
  - name: Cache node modules
    uses: actions/cache@v3   ❶ Enable cache
    with:
      path: ~/.npm    ❷ Cache rules
      key:
        ${runner.os}-${hashFiles('**/package-lock.json')}
              ❸ Cache key
      restore-keys: |
        ${runner.os}-${hashFiles('**/package-lock.json')}
        ${runner.os}

# GitLab CI Cache Example
job A:
  cache:
    key: keyA  ❹ Cache key
    paths:
      - vendor/    ❺ Cache rules

# TravisCI Cache Example
language: python
cache: pip  ❻ Implicit cache key
```

Listing 1: Cache examples in CI services.

Passive Cache. Passive cache means a CI runner does not actively clear the files generated during its CI task execution and all subsequent CI tasks will automatically reuse these files. Specifically, developers cannot explicitly specify which files should be cached or which tasks can reuse their cached files by configuration. TeamCity and Jenkins use passive cache and both are local storage.

2.3. CI Cache Matching and Inheritance

Cache Matching. All proactive caches use a key-based cache matching method. The cache key, which is generally *explicitly* specified by developers, can be hard-coded or with variables. Hard-coded keys (e.g., `keyA` in ❹ in Listing 1) do not change synchronously with code updating or operating environment switching. Alternatively, developers can add variables in a cache key. As shown in ❸ in Listing 1, there are two variables `${runner.os}` and `${hashFiles('**/package-lock.json')}`. The first variable identifies the operating system of the runner host; the second one is the hash value of the file `package-lock.json` in the repository. Therefore, the concatenated cache key remains unchanged only if the CI task runs on a runner host with the same OS and `package-lock.json` has not been modified. Moreover, some CIPs support *implicit* cache keys, which are predefined by CIPs. For example, TravisCI supports using the keyword "`cache: pip`" (❻ in Listing 1) to quickly configure the cache of Python dependent packages (i.e., `~/.cache/pip`).

When restoring cached files, the CI service will query whether the restore-key matches an existing cache object (i.e., cache hit). If all restore-keys do not match any existing cache objects (i.e., cache miss), the CI service will generate a new cache object and map it with the task's cache key.

Cache Inheritance. Some CIPs adopt a cache inheritance strategy to enhance task execution efficiency: a forked repository might inherit its parent repository's cache. For example, GitHub Actions enforces cache isolation based on branch. However, it allows a pull request to inherit caches created in the base branch or the default branch (e.g., the main branch) [15]. TravisCI also adopts a similar strategy, allowing a pull request from a forked repository without its own cache to inherit the parent's cache. Specifically, TravisCI first checks the cache of the pull request's target branch, then the parent repository's default branch cache [16]. For TeamCity and Jenkins which use passive cache, they only isolate the cache based on runner host. Therefore, different branches and even different repositories share the same cache on one runner host.

3. CI Cache Implementation Demystified

Understanding the implementation details of the CI cache is necessary to investigate its potential security threats. However, the CI cache implementation on most CIPs is not well documented. To demystify CI cache implementation, we first carefully analyze CI cache documents and the source code of several open-source CIP cache implementations (e.g., GitHub Actions). Then, we conduct black-box testing using on-premise deployed CIP instances (i.e., deploy the CIPs on our own machines) on seven CIPs. We use mitmproxy [17] to analyze network traffic generated during the execution of a CI task. To investigate the cache storage, we set up CIPs that support custom cache storage providers, using our own cloud storage (rented from AWS S3).

For the testing, we first create several private repositories with one CHP account (as the repositories' owner). For each repository, we set up a protected branch and a non-protected branch. We further fork these repositories using

TABLE 1: Overview of CI cache implementations. ○ denotes Passive Cache. ● denotes Proactive Cache. ✓ indicates the cache can be shared among different entities.

CIPs	Mode	Cache Sharing Among			
		Jobs	Branches	Repo$_{fork}$	Repo$_{other}$
GitHub Actions	●	✓		✓	
GitLab CI	●	✓	✓*	✓*	
Bitbucket Pipelines	●	✓	✓		
CircleCI	●	✓	✓	✓*	
TravisCI	●	✓	✓†	✓	
Jenkins	○	✓	✓	✓	✓‡
TeamCity	○	✓	✓	✓	✓‡

* Require user configuration (not shared by default).
† Only supports one-way sharing from the default branch to other branches.
‡ Running on the same runner host.

different CHP accounts (as normal developers). For all branches, we create the same CI tasks using the same configuration file: each CI task contains two jobs, with cache enabled using a hard-coded cache key. We also set two execution triggers for each CI task: a push event and a pull request event from the forked repository. This setup allows us to test the cache sharing/inheritance mechanisms across jobs/branches/repositories.

Specifically, we run four rounds of testing. In the first round, we initiate a code commit on the protected branch to trigger a CI task and test whether the same cache is shared by two jobs of the CI task (i.e., whether the later-executed job can use the cached files stored by the earlier-executed job). In the second round, we initiate a code commit on the non-protected branch to trigger a CI task and test whether the same cache is shared across different branches (i.e., the jobs of the CI task from the non-protected branch can use the cached files of the protected branch). In the third round, we initiate a pull request from the forked repository to the parent repository and observe whether the CI task inherits the parent repository's cache. In the fourth round, we initiate a code commit to another independent repository and record whether it shares the cached files of previous repositories on the same CI runner host. The overall features of different CIPs are shown in Table 1.

3.1. Cache Storage

Proactive Distributed Cache on CIP-Owned Cloud Storage. GitHub Actions and Bitbucket Pipelines use proactive distributed cache, uploading compressed cached files to GitHub/Bitbucket-owned cloud storage. When a CI task starts running, the two CIPs automatically generate a one-time T_{cache} for the authentication of file transmission. The token will expire immediately when the CI task ends. Moreover, the two CIPs both adopt the write-once-read-many (WORM) model [18] to protect the cache objects: once a cache key is generated, its mapped cache object cannot be changed [15]. Furthermore, the cache validity period of the two CIPs is 7 days. Developers can also delete cache objects manually before they expire.

Proactive Distributed Cache on Third-Party Cloud Storage. For proactive distributed cache, GitLab CI, CircleCI, and TravisCI use third-party cloud services (e.g., AWS S3) as their cache storage.

As authentication is required for accessing S3, both GitLab CI and TravisCI use AWS S3 presigned URLs [19] as T_{cache}. Per AWS's documents [20], there is no restriction on what files can be uploaded to a presigned URL. Anyone who knows a presigned URL can obtain temporary authorization to access the designated file in S3. Specifically, GitLab CI and TravisCI pre-allocate a storage location on S3 for cached files associated with a given CI task. Then, presigned URLs are generated and sent to a CI runner for downloading and uploading the cache object. The validity period of T_{cache} is 1 hour for GitLab CI and 2 hours for TravisCI.

For CircleCI, it creates *AWS Temporary Security Credentials* [21] as T_{cache} for the authentication. The token validity period is 15 minutes. CircleCI's document states that "Cache is immutable on write. Once a cache is written for a specific key, <redacted>, it cannot be written to again" [3].

Local Cache Storage. TeamCity and Jenkins adopt passive local cache storage by default. During a CI task, both do not actively clear the generated files. Thus, subsequent CI tasks will automatically reuse the files left by the previous CI tasks. Also, developers cannot explicitly specify cached files in the CI configuration file. Besides, Jenkins provides a plugin supporting proactive distributed cache storage (i.e., Job Cacher [22]), which can upload cache objects to AWS S3. However, the number of users of this plugin is small (merely 1,000 installs per month [22]), compared with millions of Jenkins users [23]. Thus, this paper focuses on analyzing Jenkins' passive local cache mode.

3.2. Cache Isolation and Sharing

From our four-round experiments, we find all seven CIPs do not implement cache isolation at the job level. The caches are shared among different tasks and different jobs of the same branch of a repository. Moreover, we find that these CIPs, except GitHub Actions, allow different branches of a repository to share the same cache. Among them, Bitbucket Pipelines, CircleCI, TeamCity, and Jenkins enable this feature by default. For GitLab CI, it only supports sharing cache among non-protected branches by default, and considers the cache isolation between protected and non-protected branches as a security feature [24]. Users need to manually enable cache sharing between non-protected and protected branches. Moreover, TravisCI also requires users to actively enable sharing cache among branches. Finally, we find that TeamCity and Jenkins apply the runner host-level cache isolation strategy. Different repositories running on the same runner share the same cache.

3.3. Cache Inheritance

We find that, all CIPs except Bitbucket Pipelines implement a cache inheritance mechanism. A CI task triggered by a forked repository will inherit the cache object of its parent repository unless there is an existing self-generated cache object. GitHub Actions and TravisCI implement the above inheritance by default. Since TeamCity and Jenkins adopt passive cache mode and share cache at the host level, obviously, they have the same inheritance mechanism when the forked and parent repositories run on the same runner host. For GitLab CI, it supports two execution modes [25] for a CI task triggered by a pull request from the forked repository: (1) running the task in the forked repository; (2) running the task in its parent repository. In the latter mode, the task can share the cache generated by the non-protected branch of the parent repository. For CircleCI, developers can modify the settings to allow cache sharing (i.e., inheritance) between the parent repository and all its forked builds [26].

3.4. Other Characteristics: Plugins

GitHub Actions and CircleCI support CI plugins [27], [28], which represent a set of configurable and reusable packages that can be employed in any repository. In GitHub Actions, there is an official plugin (i.e., `actions/cache` [29]) and some third-party plugins[1] that provide the cache function. As these third-party plugins are also developed based on GitHub's cache function, their cache storage locations and authentication methods are the same as the official one's. Developers can reference the official or any third-party cache plugins in the CI configuration file to use the cache function in GitHub Actions.

Meanwhile, CI plugins can be parameterized with different options. We find that some plugins take `secrets` as input parameters, which can be stored in files during plugin execution. For example, the `docker/login-action` plugin [31] from GitHub Actions read Docker Hub login credentials as the input parameters and store them in the file `~/.docker/config.json`.

4. Threat and Security Model

4.1. Threat Model

We consider a typical CI scenario adopted by an organization. The organization utilizes CHPs to maintain multiple repositories. CI is used for software development, with cache enabled. In this organization, employees have different permissions and authorization-levels for each repository. For example, repository *owners* have full access to their repositories. Collaborators have read (e.g., browse and create pull requests) and write (e.g., edit source code, create git tags, and merge pull requests) permissions, but cannot edit repository settings. The organization uses `secrets` to manage sensitive data used in CI tasks, as suggested by all CIPs. The scenario is consistent with a recent research paper on CI security [8].

We consider threats from normal users with limited permissions or low authorization-levels. They attempt to gain unauthorized access to resources (e.g., `secrets`) or to distribute malicious code stealthily (e.g., injecting backdoors into the software built by CI tasks) by exploiting implementation flaws in CI cache. Thus, we assume all stakeholders (i.e., CHPs, CIPs, and TPSs) and their communication channels are secured. We present several practical examples of adversaries and their capabilities below:

- **Unethical pull request initiator.** Adversaries can submit a pull request to execute code in the CI task of the targeted repository. This assumption does not require the code to be merged into the victim repository, since CI tasks might be configured to be automatically triggered upon receiving a pull request, but prior to the code merging [32].
- **Unethical repository collaborator.** Adversaries may be collaborators on a specific repository within the victim organization, with limited access (e.g., read and write) only to the unprotected branch of the repository. They can modify CI tasks of the authorized repository, such as build scripts or unit test code. However, they are not authorized to access protected branches, use `secrets`, or manipulate software releases. They also have no permission to other repositories in the organization. Particularly, we consider the capabilities of this type of adversary based on the default settings in CHPs. By default, in GitHub, GitLab, and Bitbucket, a collaborator has the ability to create git tags [33]. Thus, we consider an unethical repository collaborator has the capability to create git tags. Nevertheless, GitHub and GitLab provide additional tag protection functionalities [34], [35]: if the repository owner has manually configured tag protection rules, collaborators without permissions may be unable to create a specific protected tag.
- **Unethical repository owner.** Adversaries gain ownership of a single repository within the victim organization and have full access to that repository. They can also modify CI tasks of their own repository. However, they do not have the necessary permissions to access other repositories belonging to the victim organization.

4.2. Security Model of CI Cache

There is no unified and standardized security model in CI services. Thus, we have extracted a basic security model through a thorough examination of relevant open source components and careful analysis of documentation.

We first define the permission level of a cache job according to (1) whether it has the permission to access secrets, (2) the permission level of the CI task initiator it belongs to, and (3) whether it is running on a protected branch. For example, a protected branch can usually only be operated by trusted developers, while a non-protected branch can be operated by any repository member. Similarly, the parent repository can only be operated by repository members, while the forked repository can be operated by any developer with read permission of the parent repository. The specific define rules are as follows:

- **Rule 1.** In the same repository, a job that can access CI secrets has higher permission than a job that cannot do it.
- **Rule 2.** In the same repository, a job of the protected branch has higher permission than a job of the non-protected branch.

[1]For example, Swatinem/rust-cache [30], a popular Rust cache plugin.

- **Rule 3.** A job of the parent repository has higher permission than a job triggered by a pull request initiated by the forked repository. This is because CIPs typically consider the latter is initiated by less trusted users [36], [37].

Moreover, we mark a CI job that generates a cache object as the source job and a CI job that uses a cache object as the sink job. Source job and sink job can be different jobs of (1) the same task, (2) different branches, or even (3) different repositories.

5. Identified Threats in CI Cache

Combining our proposed threat model and the extracted security model, we present several potential security threats that may arise from cache usage in CIPs. Through these attacks, adversaries can access unauthorized resources, steal secrets, and inject malicious code into software artifacts.

5.1. Improper Write-Up

There is a risk of cache poisoning if a cache object sourced from (i.e., written by) a low-permission job is used by a high-permission job. Once attackers have poisoned the cache of the victim repository, malicious code may be embedded during software building. Attackers can also exploit the cache to tamper with dependent packages of the unit test code. They can covertly run malicious code to steal tokens and/or manipulate CI tasks' results. Such a threat is hard to detect because attackers do not pollute the unit test code directly.

5.1.1. Cache Poisoning Attack

The cache poisoning attack can be divided into three categories based on different source/sink jobs. We consider the attack with high risk as it enables cross-repository threats.
Cross Repository Cache Poisoning. TeamCity and Jenkins adopt a runner host-level cache isolation strategy: repositories running on the same runner host share the same cache. Therefore, attackers (e.g., unethical pull request initiator or unethical repository collaborator) can trigger a CI task of a repository (with low-level permissions) to inject malicious code into cached files (e.g., dependent packages). The malicious code will affect victim repositories on the same runner host if the cached files are used by their CI tasks.
Cross Branch Cache Poisoning. Except for GitHub Actions, all other six CIPs allow different branches of the same repository to share the cache (GitLab CI and TravisCI require user configuration as described in Section 3.2), making them vulnerable to this type of attack. A typical attack scenario is that attackers (e.g., unethical repository collaborators) only have permission to a non-protected branch of the victim repository. They can then initiate a CI task on the non-protected branch and generate a malicious cache object. Because the cache is shared across branches, the poisoned cache object might be used even by a protected branch of the victim repository.
Cross Job Cache Poisoning. We find that GitHub Actions and GitLab CI can grant different levels of permissions to different jobs of the same branch. For example, they can restrict access to secrets to specific jobs (e.g., with high-level permissions). However, attackers (e.g., unethical repository collaborators) can bypass this restriction even with low-level permissions only, because GitHub Actions and GitLab CI share the same cache for all jobs of the same branch. Attackers can exploit this vulnerability to poison the cache generated by a low-level permission job (e.g., a unit test job) and then steal secrets from high-level permission jobs (e.g., a release job) of the victim branch.

Note that popular package managers, such as `npm` and `pip`, cannot defend against cache poisoning attacks through their dependency lock and checksum verification [38], [39] mechanisms. The dependency lock only ensures that dependent packages use a specific version; and checksum verification only verifies the integrity of the package file downloaded from the package registries. Both mechanisms cannot detect the case that attackers directly modify the content of the package file (i.e., cached file) on the host machine.

5.2. Improper Read-Up

There is a risk of data leakage when a cache sourced from a high-permission job is used (e.g., read) by a low-permission job. For instance, if the high-permission job mistakenly caches a file containing sensitive information (e.g., API tokens), the low-permission job can then access the cache, read the file and potentially steal the tokens from the victim. We find that all seven CIPs suffer from this risk.

One important data that may be leaked from the cache is `secrets`, which is also a significant concern of CI users [40]. We take a GitHub repository (i.e., `openzipkin/zipkin-dependencies` [41]) as an example. This repository will publish a popular docker image (over 10M downloads) with the same name [42] to Docker Hub using its custom CI task of GitHub Actions. As Listing 3 shows, before executing the "`docker push`" command (line 12), the CI task will execute the "`docker login`" command (line 11) to log in to Docker Hub. The username and token used for login are passed by CI secrets (line 14 and 15), which is the recommended way in GitHub Actions [11]. After successfully logging in, the "`docker login`" command will store the username and token (both in plaintext) in the file `~/.docker/config.json` [43]. However, according to the cache rule (line 5), the task will cache all files under the folder `~/.docker/`, including the file containing docker login credentials. Thus, attackers can access the cache, steal the credentials, and further perform other malicious activities (e.g., push malicious images to Docker Hub using the victim's account).

5.2.1. Cache Leakage Attack

Unethical pull request initiators or unethical repository collaborators can launch high-risk cache leakage attacks, which could potentially leak *secrets*. Particularly, we find that cache leakage might be caused by different stakeholders in multiple situations.

Official Cache Service with Vulnerable Custom Cache Rules. Developers often use CIP's official cache service, but they might mistakenly add paths including sensitive files to the cache rules. One vulnerable example is the above-mentioned `openzipkin/zipkin-dependencies`. As another example, a popular database management tool (with 100M+ downloads), *liquibase* [44], has this type of issue too. Its CI configuration file (as shown in Listing 2) shows it uses GitHub Actions' official cache service (i.e., actions/cache). During the execution of its CI task, the tool's software signing certificates and their passwords will be passed in via CI `secrets`, and then written into two files[2,3]. However, according to the custom cache rules (i.e., `./**/target`), both files will be cached. If the cache is leaked and accessed by other users, it will potentially cause severe damage.

```
1  name: Package Artifacts
2  steps:
3    - name: Built Code Cache
4      uses: actions/cache@v3.3.1
5      with:
6        key: built-code-${{github.run_number}}-${{
7              github.run_attempt}}
8        path: ./**/target    ⇐ Cache rule
9    - name: Build & Sign Artifacts
10     env:
11       INSTALL4J_APPLE_KEY:
             ${{secrets.INSTALL4J_APPLE_KEY}}
12       INSTALL4J_APPLE_KEY_PASSWORD:
             ${{secrets.INSTALL4J_APPLE_KEY_PASSWORD}}  ⇐
             Define secret variables
13       INSTALL4J_WINDOWS_KEY:
             ${{secrets.INSTALL4J_WINDOWS_KEY}}
14       INSTALL4J_WINDOWS_KEY_PASSWORD:
             ${{secrets.INSTALL4J_WINDOWS_KEY_PASSWORD}}
15     run: |
16       mkdir -p liquibase-dist/target/keys
17       echo "Saving apple key"
18       echo "$INSTALL4J_APPLE_KEY" | base64 -d >
             liquibase-dist/target/keys/datical_apple.p12
             ⇐ Write a secret variable to file
19       echo "Saving windows key"
20       echo "$INSTALL4J_WINDOWS_KEY" | base64 -d >
             liquibase-dist/target/keys/datical_windows.pfx
              ⇐ Write a secret variable to
             file
```

Listing 2: The CI configuration file (i.e., create-release.yml) of liquibase/liquibase, which writes secret variables (i.e., software signing keys) to the files via shell commands (line 18 and 20).

Vulnerable Official Cache Templates. Many CIPs provide pre-configured CI cache templates (e.g., predefined keywords). If these templates' cache rules include sensitive files, developers may inadvertently and unknowingly expose sensitive data. After carefully scrutinizing CIPs' official cache templates, we find that multiple templates are misconfigured and their users potentially suffer from serious data leakage risks. We list all vulnerable official cache templates in Table 2: the Gradle templates of GitLab CI and CircleCI, the Maven templates of CircleCI and TravisCI, and the Cargo template of TravisCI are vulnerable to this threat. Particularly, these templates by default cache all files in the package managers' configuration folders, which include configuration files that contain sensitive data. For example, Maven's configuration folder (i.e., `~/.m2/`) contains a subfolder (i.e. `~/.m2/repository/`) for storing dependent packages and a configuration file (i.e. `~/.m2/settings.xml`) that may store Maven credentials. However, both CircleCI and TravisCI cache all files in the configuration folder, potentially causing credentials leakage.

```
steps:
  - name: Cache Docker
    uses: actions/cache@v2
    with:
      path: ~/.docker  ⇐ Cache rule.
      key:${{runner.os}}-docker-${{hashFiles('**/Dockerfile')}}
      restore-keys: ${{runner.os}}-docker
  - name: Docker Push
    run: |
      build-bin/git/login_git &&
      build-bin/docker/configure_docker_push &&  ⇐
          Executes docker login command.
      build-bin/docker_push  ⇐ Executes docker push
          command.
    env:
      DOCKERHUB_USER: ${{secrets.DOCKERHUB_USER}}
      DOCKERHUB_TOKEN: ${{secrets.DOCKERHUB_TOKEN}}
```

Listing 3: The CI configuration file (i.e., docker_push.yml) of openzipkin/zipkin-dependencies.

Vulnerable Third-Party Cache Plugins. Except for the official cache service, there are third-party cache plugins in GitHub Actions. These plugins may also provide pre-configured cache rules to simplify configuration. Similarly, if their rules are misconfigured, sensitive files may be cached without developers' awareness. For example, a popular third-party cache plugin, `Swatinem/rust-cache`, which is used by more than 2,800 repositories, provides preconfigured cache rules (i.e., `~/.cargo/`). For some Rust-related CI tasks, they will execute the command `"cargo login"` to log into the Cargo registry for publishing packages. Unfortunately, the command will write the login token (in plaintext) into the file `~/.cargo/credentials` [52], and then the file will be cached according to the plugin's rules, potentially leading to token leakage. A real vulnerable example is a popular package, *jsonc-parser* [53], with more than 640K downloads in Cargo registry. Once attackers steal its token, they can inject malicious code into the package, causing a serious software supply chain threat.

5.3. Improper Token Permission

For CIPs adopting distributed cache storage mode, they all use a token (i.e., T_{cache}) to authorize and control cache access. To avoid potential privilege escalation, T_{cache} should be limited to the cache of the originated repository, without any permissions to caches of any other repositories. Moreover, if a CIP's cache is designed as read-only, its T_{cache} should

[2] `liquibase-dist/target/keys/datical_apple.p12`
[3] `liquibase-dist/target/keys/datical_windows.pfx`

TABLE 2: Vulnerable official cache templates.

CIPs	Templates	Cache Rules	Sensitive Files	Sensitive Data
GitLab CI	Gradle [45][*]	`.gradle/`	`.gradle/gradle.properties`	Project configuration properties, e.g., registry credentials [46].
CircleCI	Maven [47]	`~/.m2/`	`~/.m2/settings.xml`	Configurations of the Maven tool, e.g., authentication secrets [48].
CircleCI	Gradle [49]	`~/.gradle`	`~/.gradle/gradle.properties`	Project configuration properties, e.g., registry credentials [46].
TravisCI	Maven [50]	`~/.m2/`	`~/.m2/settings.xml`	Configurations of the Maven tool, e.g., authentication secrets [48].
TravisCI	Cargo [51]	`~/.cargo/`	`~/.cargo/credentials`	The API token of Cargo registry used to publish packages [52].

[*] Although GitLab CI only allows caching files under the current directory, there is still a risk in the Gradle official cache template because it changes the Gradle home directory (i.e., `GRADLE_USER_HOME="$(pwd)/.gradle"`) [45].

not have permission to modify cache objects. However, we find that TravisCI and CircleCI have token permission issues. In both CIPs, the T_{cache} for pull requests has permission to modify the parent repository's cache objects.

5.3.1. Cache Privilege Escalation Attack

If over-privileged T_{cache} is leaked (e.g., via the attacks proposed in [8]), attackers (e.g., unethical pull request initiators) can exploit it to escalate their privileges and cause damage beyond expectations (high-risk).

TravisCI. Since TravisCI uses AWS S3 presigned URLs as T_{cache}, anyone who obtains the presigned URL can temporarily access specific files in S3. Meanwhile, TravisCI adopts a three-level cache inheritance strategy for a pull request from a forked repository. It will check caches in sequence: (1) the pull request cache, (2) the pull request's target branch cache (in the parent repository), and (3) the parent repository's default branch cache. We find that a CI task triggered by a pull request from a forked repository always generates all three presigned URLs for the above possible caches. Moreover, all of these presigned URLs will be passed to the CI runner and then written to a file (i.e., `~/.casher/bin/casher`) in the runner host. The runner checks the existence of caches in order and restores the first downloadable cache to the local host. However, by adding code in the CI configuration file or into the codebase (e.g., unit test code) used in a CI task and then initiating a pull request from the forked repository, attackers can easily access the file, obtain the three presigned URLs, and further illegally manipulate (e.g., read and/or write) the parent repository's default and target branch caches.

CircleCI. CircleCI uses *AWS temporary security credentials* [21] as T_{cache} for the authentication of cache objects uploading and downloading. Since CircleCI saves T_{cache} in the CI runner's memory and the runner runs with root privileges [54], attackers can adopt similar attacking methods in TravisCI, i.e., adding code in the CI configuration file or into the codebase, then they can steal the T_{cache} directly from the runner process by reading its memory.

Even worse, CircleCI's T_{cache} is over-privileged. Although CircleCI explicitly states that its cache is immutable on write [3], we find that CircleCI's T_{cache} has permission to modify cache objects. Furthermore, we find that its T_{cache} can traverse all the caches of the victim repository to read/write arbitrarily without any limitation. Thus, attackers can initiate a pull request from the forked repository to the victim repository, steal T_{cache}, and then arbitrarily modify cache objects of any branch of the victim repository.

5.4. Improper Cache Eviction

We find that some CIPs do not evict cache objects properly when the permission of a repository member has changed. Specifically, even if users have lost their access privileges to a repository (e.g., fired employees), their cache objects are still valid and might be used by the repository. This improper cache eviction enables attackers to continue attacking a victim repository (e.g., implant a backdoor in the cache) even after losing their access privileges.

We identify three practical scenarios where untimely cache eviction can cause security threats. (1) User permission revocation: when a user's permission is revoked, cache objects created by this user are not promptly deleted. (2) `git tag` deletion: a tagged cache is not timely deleted when the corresponding tag is deleted. Particularly, `git tag` is a crucial feature commonly employed for marking new release versions. GitHub Actions provides a distinct cache isolation strategy for each `git tag` [55]: a cache generated by a specific `git tag` can only be shared and used by the same `git tag`. However, tags can be created, deleted, and then recreated. We find that the cache associated with a tag remains intact even after the tag is deleted. Therefore, an attacker can create multiple tags to generate multiple caches and subsequently delete the tags before losing the access permission. When subsequent benign developers recreate any of the above tags, the triggered CI task will use the poisoned cache. (3) Repository ownership transfer. All the three CHPs provide the repository transfer function: transferring the ownership of a repository to another user. To avoid path collision and unintended cache sharing among different repositories, some CIPs use *repository id* (from CHPs) to generate the object path in the cache storage (as shown in Figure 1). However, the *repository id* remains unique and unchanged in each CHP, even though the repository's owner has been changed. This mechanism will cause serious threats if CIPs do not invalidate the cache generated by the old owner: the new owner will automatically and unknowingly use the cache generated by the old owner when running a CI task. Unfortunately, we find GitHub Actions, GitLab CI, CircleCI, and TravisCI are all vulnerable to this threat.

5.4.1. Cache Backdoor Attack

We propose a new attack exploiting the above mechanism.

Threat Model. The threat model is slightly different from our general model. We assume that at first an attacker has write access to a repository (e.g., as a repository owner or collaborator), which does not contain malicious code.

Later the attacker loses access to the victim repository. This scenario is not rare: for example, the attacker is a disgruntled employee who is leaving an organization.

Attackers can implant malicious code (e.g., a backdoor) in the cache (e.g., a dependent package), while the original repository looks perfectly benign. After the attacker loses access privileges (e.g., repository ownership transfer), because the attacker-generated cache objects are not invalidated by CIPs, the backdoor will be injected into the artifact when the new user (i.e., victim) executes a CI task of the repository. Such a threat will not disappear until the poisoned cached file expires, is deleted, or is overwritten by a benign one.

We find that the above mentioned attack can pose a threat to a large number of repositories, including many repositories from large organizations. However, as attackers must have at least collaborator permissions and the attack time windows is limited, we consider this attack as low-risk. We present a detailed analysis in Section 7.4.

6. Measurement

To investigate the current security status of the cache usage in the open source community, we design *CAnalyzer* to conduct a large-scale measurement on the three mainstream CHPs. Overall, *CAnalyzer* first collects repositories with CI-enabled from CHPs. Then, *CAnalyzer* distills various properties of each repository, such as CI cache rules and secrets usage. Finally, based on the risks identified in this paper, *CAnalyzer* locates affected repositories.

6.1. Data Collection

To obtain a comprehensive set of repositories with various combinations of CIPs and CHPs, we adopt multiple strategies for data collection. First, all CIPs except TeamCity support configuring CI via configuration files. Thus, we identify a repository's CIP by examining its CI configuration file and exclude TeamCity from our measurement. Second, for repositories on GitLab and Bitbucket, we retrieve all publicly available repositories and their contents through CHPs' public APIs [56], [57], [58], [59]. Third, in the case of repositories on GitHub, we extract repositories using GitHub Actions from the GHArchive data [60] by analyzing `github_bot` events, similar to previous works [4], [8]. In addition, we utilize *GitHub Activity Data* on BigQuery [61] to gather information on repositories that employ other CIPs. Overall, the counts of collected repositories with different CIPs enabled are: GitHub Actions (683,125), GitLab CI (306,159), Bitbucket Pipelines (29,169), CircleCI (123,075), TravisCI (554,961), and Jenkins (19,798). The data was collected on January 2023.

6.2. Basic Threat Analysis

CAnalyzer detects potential threats according to the predefined rules, as detailed below.

Repository Parser. First, we analyze a repository's CI configuration file to identify the CI service used by the repository, the cache service (if used), and the usage of CI secrets. Second, for repositories hosted on GitHub, we obtain the branch protection settings of public repositories through the GitHub API [62], and analyze their events in the GHArchive historical data [60]. Specifically, we extract `"repository id"` (i.e., the unique/unchanged identifier of a repository) and `"user id"` of the repository owner (i.e., the unique/unchanged identifier of a GitHub user [63]) from these events. By analyzing the change of the historical owners' user ids of a repository, we can identify whether the owner of the repository has been transferred between different users. Similarly, we extract *MemberEvent* from the GHArchive data to identify all historical collaborators in each repository. Due to permission restrictions [64], we are unable to obtain the current list of collaborators for a repository using the GitHub API. Fortunately, we find that the pull request data of a repository includes the current role of its initiator (i.e., whether a user is currently a collaborator), so we collect pull requests through the GitHub API [65] to filter out users who are currently collaborators.

Cache Parser. *CAnalyzer* extracts all used cache keys from the CI configuration files. We track the data flow of configuration-related variable references to ensure correct cache key-object mappings. Then, the cache key is used as an identifier to locate the cache source job and sink job. Moreover, we also analyze how `secrets` (if used) are referenced in each job.

Vulnerability Detection. Based on the analysis in Section 5.3.1, we consider repositories using TravisCI or CircleCI's cache feature to be vulnerable to the *Cache Privilege Escalation Attack*.

For *Cache Poisoning Attack*, there are three types of threats. We consider: (1) repositories that use Jenkins have the risk of cross repository cache poisoning; (2) repositories using CircleCI's cache feature and sharing the same cache key among different branches have the risk of cross branch cache poisoning (we cannot accurately detect vulnerable repositories using other CIPs, as discussed in Section 7.3); (3) repositories using GitHub Actions or GitLab CI and sharing the same cache key among different permission-level jobs in the same branch have the risk of cross jobs cache poisoning.

For *Cache Backdoor Attack*, we assess the potential impact of (1) unethical collaborators by analyzing the collaborator removal events in popular repositories; (2) tagged cache reuse by analyzing the `git tag` trigger events; (3) ownership transfer based on the number of repository transfer events, especially the transfer from a personal account to an organizational account.

6.3. Data Leakage Analysis

The usage of secrets is complicated, simple attribute extraction and CI configuration file analysis cannot accurately identify data leakage problems caused by the cache. Therefore, we design and implement a specialized data leakage analysis module. Overall, we first identify all possible secrets related variables, then further locate sink files that may contain secrets. Finally, *CAnalyzer* detects data leakage according to whether these sink files are cached in an either explicit or implicit way.

Secret Variable Identification. All CIPs support secret variables and users can reference them in CI configuration files. For example, GitHub Actions adopts `${{secrets.SECRET_NAME}}` format, while others adopt formats like `${SECRET_NAME}`. Thus, we extract secret variables by parsing their corresponding formats. *CAnalyzer* further filters out non-secret variables (i.e., variables declared in CI configuration files but with literal values) as well as CIP's predefined variables. The remaining variables are considered secret variables.

Second, all the above CIPs use yaml data language [66] in their CI configuration files. Thus, by parsing these files, *CAnalyzer* extracts the `Mapping` operator, which can map a scalar (i.e., secrets) to another scalar (i.e., a variable derived from secrets). Then *CAnalyzer* identifies mapping operations and further locates variables related to secrets.

Secrets Sink File Tracing. Next, we attempt to identify files that secrets might be written to. After some manual analysis, we focus on three commonly used patterns of secret variables: (1) Used directly as parameters to shell commands in the CI configuration file (e.g., `echo $SECRET > file`). (2) Used by other files in the repository (e.g., script files), which are invoked in the CI configuration file with secret variables as parameters. (3) For GitHub Actions and CircleCI, secret variables may also be referenced by third-party plugins.

If a command reads secrets and explicitly writes them to a file (e.g., using redirection operators such as >, >>), we consider the output file as sensitive. For implicit writing caused by shell tools (e.g., the command "`cargo login $TOKEN`"), we employ a semi-automated approach to test whether these tools will potentially write sensitive data to files. First, we extract all candidate shell tools in CI tasks that take secrets as input. We also record their corresponding shell commands. Second, since a shell tool will likely generate sensitive files during specific command executions, we generate a set of call templates by parsing the parameters of the shell commands. For instance, for a concrete shell command such as `docker login -u $USERNAME -p $PASSWORD`, we derive the corresponding template as `docker login -u $1 -p $2`. *CAnalyzer* then automatically conducts tests on candidate shell tools utilizing the generated templates. To precisely monitor generated sensitive files, we use strace [67] to dynamically trace and record all file write operations during the execution. After the test, *CAnalyzer* further analyze the contents of the recorded files to determine if they contain any sensitive value. Specifically, we first search plaintext secrets in all files, and further use TruffleHog [68], an open-source credentials detection tool, to scan the files for any sensitive data. In this way, we can effectively identify sensitive data derived from secrets. For instance, *CAnalyzer* can handle the case where the OAuth token is obtained through username/password and stored in a file. If sensitive values are detected within any files, we consider that the shell tool has the potential to generate sensitive files under the corresponding template.

It should be noted that many test templates need valid input for successful execution. One of the most common type of input is authentication credentials for cloud services. For example, to execute the command `az login -u $1 -p $2`, we need to provide the correct user name (i.e., `$1`) and API token (i.e., `$2`) of the Azure cloud services. As it is difficult to automate this step, we manually identify 43 cloud services from 1,835 templates of the top 100 most widely-used shell tools. Subsequently, we manually create accounts for these services and generate necessary parameters, such as API tokens, to ensure the successful execution of their test templates.

Similarly, for the case where secrets are referenced by third-party plugins, we extract all plugins that take secret variables as input parameters and generate test templates. However, unlike shell tools, plugins cannot run independently without the CI execution context. Thus, we generate a CI task for each test template and execute the CI task using our own self-hosted runner. During the execution, we employ kprobes [69] to monitor the files created throughout the process. Then, we check whether sensitive files are generated after the execution. Similar to shell tools, plugins may also require parameters for successful executions. Therefore, we manually generate the requisite parameters for testing the top 100 most widely-used plugins in our collected dataset. Based on the results, we summarize the rules (shown in Section 7.1) for detecting secrets sink files.

Data Leakage Detection. Only when secrets sink files are cached, there is a risk of secret data leakage. Thus, *CAnalyzer* uses explicit cache analysis and implicit cache analysis to extract the cache rules in CI tasks. Then, *CAnalyzer* conducts cache reachability analysis to verify whether secrets sink files are stored in the cache.

For explicit cache, developers explicitly specify which files should be cached by cache rules in the CI configuration file (as mentioned in Section 2.2), thus we simply parse the rules following each CIP's documents [70], [71], [72], [73] and check whether a sensitive file is matched by cache rules (i.e., stored in cache objects).

For implicit cache (i.e., a repository's CI configuration file does not contain any cache rules directly, but it does contain a cache plugin with built-in cache rules), we find that such implicit cache exists in GitHub Actions, Bitbucket Pipelines, and TravisCI. For Bitbucket Pipelines and TravisCI, their implicit cache rules are listed in the documents. For example, in Bitbucket Pipelines, `cache: maven` implicitly specifies the cache rule of `~/.m2/repository`. We thus parse each cache template following its document. For GitHub Actions, it supports plugins: both the official and third-party plugins can read and write cache. Thus, we extract all plugins from the collected repositories using GitHub Actions and then download their source code. If a plugin's source code contains `@actions/cache` (the interface to import cache), we consider the plugin with cache function and further extract the candidate cache rules automatically from its code. Finally, we manually confirm its implicit cache rules.

Cache Reachability Analysis. There are two special scenarios that can prevent sensitive files from being cached, even if the files have been specified by cache rules. The first scenario involves the creation of a sensitive file within some specific callbacks. For example, scripts executed in the `after_deploy` phase of TravisCI occur after the cache saving [74], so files generated during this phase will not be stored in the cache. A concrete example is the `gluon-lang/gluon` repository [75]. We have manually checked each CIP's documentation to identify and exclude such callbacks. The second scenario is conditional caching, where the creation of sensitive files and cache saving must satisfy specific conditions. If these two conditions are mutually exclusive, there is no security risk. A real case can be observed in the `dprint/dprint-plugin-typescript` repository [76]: the condition for storing cache (i.e., not cache for git tag creation) is incompatible with the condition for generating sensitive files (i.e., only when a git tag is created). If cache creation or cache saving is unconditional, or both conditions can be satisfied at the same time, we consider that a cache is reachable. We extract conditions in the cache code block and the sensitive file code block from the CI configuration file. Then we utilize pyDatalog [77] to evaluate the truth value of these conditions, and consider the cache is reachable (i.e., vulnerable) only if both conditions can be satisfied.

6.4. Ethical Considerations

All experiments are conducted in an ethical way. First, all vulnerability analyses and threat validations are done on our own repositories/projects. We do not launch attack on any vulnerable repositories. Instead, we use our own account to create fresh new repositories, and copy the CI configuration file of the vulnerable repositories for data leakage risk testing. Second, we never test vulnerable credentials in real scenarios, which could be unethical. Instead, to ensure successful execution, we have made necessary changes that do not affect the verification (e.g., replacing vulnerable credentials with our testing ones) to the CI configuration file. We also use our own resources to test/verify the related risks (e.g., privilege escalation and cache poisoning). No experiments are conducted on third-party repositories/projects or any resources that do not belong to us. Finally, after the experiments, we have timely disclosed all our findings to corresponding stakeholders (details in Section 8.2). We also follow the 90-day disclosure deadline policy [78] whereby issues can be made public after 90 days.

6.5. Limitations

Firstly, our work goes beyond open-source repositories as we also consider common organization scenarios. However, we can only measure public repositories and cannot assess the security risks of private projects, particularly, those projects inside organizations. Secondly, the measurement study reveals repositories that could be potentially vulnerable by the *default* setting, in which collaborators can create git tags. As mentioned in our threat model, in practice, collaborators cannot create a specific protected tag if the repository owner has manually set comprehensive tag protection rules However, due to permission restrictions by CHPs, we do not have access to the full settings of a repository (e.g., whether a repository has set tag protection rules is only available to repository member with admin privileges [79] in GitHub). Thus, our measurement study reveals repositories that could be potentially vulnerable by the default setting (i.e., without proper tag protection rules). But the exploitability of these "potentially vulnerable repositories" depends on their specific permission settings. Thirdly, our experiments on sink files only examined the latest versions of shell tools and CI plugins. However, in real CI tasks, various versions might be employed, potentially causing false positives/negatives. Finally, our secrets sink file analysis is partially done by manual and only covers the most popular targets (i.e., the top 100 shell tools and the top 100 CI plugins) that use `secrets`. Thus, the actual situation might be even worse than our results.

7. Measurement Results

We adopt *CAnalyzer* to understand the potential impact of cache related threats in the open source community. For all uncovered "potentially vulnerable repositories", we conduct manual verification to confirm their vulnerability based on the default setting as discussed in Section 6.5. We acknowledge that launching a successful attack on these repositories may require the attacker to acquire additional permissions or gain trust, as certain security policies might be enabled.

7.1. Cache Leakage Attack

Among the top 100 most-used shell tools, 26 of them write sensitive data into files when executing specific commands. Table 3 shows some of them. Meanwhile, 9 GitHub Actions (partially shown in Table 4) and 13 CircleCI plugins (out of the top 100 most used) read secrets as input and write secrets into files. Further analysis indicates that: (1) the most common scenario for writing secrets (e.g., API tokens and login credentials) to files is to access protected cloud services (such as Docker Hub and AWS). Many tools/plugins store authentication credentials in a specific file for future use (e.g., automatic login and authorization). Usually, this behavior is well documented in tools/plugins' manuals. (2) The second scenario is writing sensitive parameters into the configuration file. For example, `yarn config set npmAuthToken` command writes the npm authentication token as a configuration item, and `docker trust key load` command copies a private key into docker's configuration file directory. Even worse, these tools' documents typically do not describe these operations in detail, and thus users may be unaware that sensitive files are created and cached. (3) The third scenario is writing sensitive data to the tool's execution log. For example, `npx vercel --token` command writes the vercel API token in plaintext to the `~/.npm/_logs/*.log` file. Unfortunately, this behavior is usually not well-documented either. Table 7 in Appendix

TABLE 3: Top 5 shell tools used in CI tasks that create sensitive files.

Tools	Commands	Generated Sensitive Files	Sensitive Data
docker	`$ docker login`	`~/.docker/config.json`	Docker registry login credentials.
	`$ docker trust key load`	`~/.docker/trust/private/*.key`	Private key for signing/verifying Docker images.
npm	`$ npm login`	`~/.npmrc`	npm registry login credentials.
aws	`$ aws configure set aws_secret_access_key`	`~/.aws/credentials`	AWS services access key secret.
	`$ aws configure set aws_access_key_id`	`~/.aws/credentials`	AWS services access key id.
npx	`$ npx vercel --token`	`~/.npm/_logs/*.log`	Vercel services access credentials.
poetry	`$ poetry config http-basic.pypi __token__`	`~/.config/pypoetry/auth.toml`	PyPI registry login credentials.

TABLE 4: Top 5 GitHub Actions plugins used in CI tasks that create sensitive files.

Plugins	Plugin Input Parameters	Generated Sensitive Files	Sensitive Data
docker/login-action	password	`~/.docker/config.json`	Docker registry login credentials.
FirebaseExtended/action-hosting-deploy	firebaseServiceAccount	`/tmp/tmp-*.json`	Firebase services API token.
crazy-max/ghaction-import-gpg	gpg_private_key	`~/.gnupg/private-keys-v1.d/*.key`	GPG private key.
akhileshns/heroku-deploy	heroku_api_key	`~/.netrc`	Heroku services API token.
hashicorp/setup-terraform	cli_config_credentials_token	`~/.terraformrc`	Terraform services API token.

shows a comprehensive list of tools/plugins used in the CI task that create sensitive files.

Vulnerable Custom Cache Rules (DL_1). We find that 37 GitHub Actions, 12 TravisCI, and 3 CircleCI repositories use risky cache rules and will cache files containing secrets. The most commonly leaked secrets is the login credentials of the Maven registry (contained in the setting file `~/.m2/settings.xml`): 22 repositories write this file to CI caches. Among these repositories, 3 actively write secrets to the setting file using commands (e.g., "echo") and store them in the cache. Meanwhile, 19 repositories use a third-party plugin of GitHub Actions (i.e., `whelk-io/maven-settings-xml-action`) to write the "password" parameter to the file. However, the plugin's document does not clearly state that when using the plugin, the file `~/.m2/settings.xml` should not be cached. Therefore, users might be unaware of the risk of credential data leakage. Similarly, the login credentials of the Cargo registry (contained in the file `~/.cargo/credentials`) are also leaked by 16 repositories using GitHub Actions and 4 repositories using TravisCI. Moreover, we find two repositories using TravisCI leak AWS login credentials and two repositories using GitHub Actions leak `vercel` API credentials through the log files (i.e., `~/.npm/_logs/*.log`) created by the `npx vercel --token` command.

Table 5 shows some repositories at risk, including several very popular repositories. For example, `liquibase/liquibase`, a popular database management tool that has been downloaded over 100M times [80], leaks its software signing certificates. If attackers steal the credentials, they can release packages containing malicious code stealthily, potentially affecting a large number of users. We provide a Proof of Concept (PoC) repository[4] of this repository along with detailed reproduce steps. Note that this vulnerability has been confirmed and fixed by the liquibase team, who rewarded us with a gift.

Vulnerable Official Cache Templates (DL_2). We find that vulnerable official cache templates are widely used by 792 GitLab CI, 1,316 CircleCI, and 6,966 TravisCI repositories. If these repositories' CI tasks use CI secrets, they potentially leak their credentials. In the data we collected, we have identified one real repository (i.e., `chalharu/rust-nearly-eq` [81]) using TravisCI stores the file containing the secrets in the cache and leads to credential data leakage.

Vulnerable Third-Party Cache Plugins (DL_3). For GitHub Actions, we find 76 third-party cache plugins from a collection of 33,354 plugins. After manual verification, we confirm that the plugin `swatinem/rust-cache`, used by more than 2,800 different repositories, suffers from this problem. Its default cache rules (`~/.cargo/`) will store the file `~/.cargo/credentials` into the cache. We have further identified 25 repositories affected. Some of them and their corresponding packages in the Cargo registry are shown in Table 5.

7.2. Cache Privilege Escalation Attack

Repositories using TravisCI or CircleCI's cache function are potentially vulnerable to this threat. Our measurement results show that 105,757 (19.06%) repositories using TravisCI and 9,458 (7.68%) repositories using CircleCI enable the cache function.

We select representative target repositories for further analysis. We rank vulnerable repositories hosted on GitHub based on their star counts and select the top 1,000 as targets. Then we analyze their complete pull request histories using the GitHub API [82]. The results show that 570 (57.00%) of the target repositories have accepted and merged pull requests from forked repositories. These repositories include very popular repositories and repositories from large organizations like Facebook, Google, and Twitter. Table 6 shows some of the affected repositories and their cache usage (i.e., using cache for storing dependent packages and/or build intermediates).

7.3. Cache Poisoning Attack

Since Jenkins shares cache at the runner host level, all 19,798 repositories that use Jenkins are at risk of *cross-repository* cache poisoning. Note that TeamCity is also

[4]https://github.com/cicache-poc/liquibase__liquibase

TABLE 5: Some vulnerable repositories, their sensitive data at risk, and the potential consequences.

Repositories	CIPs	Types	Sensitive Data	Possible Consequences of Secrets Leakage
liquibase/liquibase	GitHub Actions	DL1	Software signing certificates	Publish a malicious version of the victim software (100M+ downloads).
openzipkin/zipkin-dependencies	GitHub Actions	DL1	Docker Hub login credentials	Publish a malicious version of the victim docker image (10M+ downloads).
dprint/jsonc-parser	GitHub Actions	DL3	Cargo registry credentials	Publish a malicious version of the victim package (664K downloads).
chalharu/rust-nearly-eq	TravisCI	DL2	Cargo registry credentials	Publish a malicious version of the victim package (89K downloads).
SBJson/SBJson	CircleCI	DL1	CocoaPods registry credentials	Publish a malicious version of the victim package (3.7K stars on GitHub).

TABLE 6: Some repositories vulnerable to the *Cache Privilege Escalation Attack* and their caches' usage. *Dep.*-Dependent Packages, *Inter.*-Build Intermediates.

Repositories (# of Stars)	CIPs	Dep.	Inter.
facebook/react-native (109K)	CircleCI	✓	✓
fastlane/fastlane (36.9K)	CircleCI	✓	✓
aria2/aria2 (30.2K)	TravisCI	✓	
pytorch/fairseq (21.9K)	CircleCI	✓	
google/TensorNetwork (1.7K)	TravisCI	✓	
twitter/bijection (0.6K)	TravisCI	✓	

vulnerable, but we are unable to collect related data, as mentioned in Section 6.1.

For *cross-branch* cache poisoning, we evaluate repositories with CircleCI on GitHub (which publicly provides branch protection data). While both CircleCI and TravisCI get integrated with GitHub and are vulnerable to cross-branch cache poisoning, cross-branch cache sharing is disabled by default in TravisCI. We find that 2,854 of 9,643 (29.60%) repositories using CircleCI on GitHub have at least one protected branch. Among them, 1,505 (52.73%) have enabled the cache function and 935 (32.76%) share the same cache between a protected branch and a non-protected branch, which are vulnerable to cross-branch cache poisoning.

For *cross-job* cache poisoning, we analyze 100,986 and 14,807 repositories with cache-enabled that use GitHub Actions and GitLab CI respectively. We find that 23,037 (22.81%) repositories using GitHub Actions and 594 (4.01%) repositories using GitLab CI have at least two different levels of permissions under the same branch jobs. Among them, 15,942 and 469 repositories share the same cache objects (i.e., using the same cache keys) in jobs with different levels of permissions respectively, which are at risk of cross-job cache poisoning.

7.4. Cache Backdoor Attack

Collaborator Removal. Collaborators can be removed from a repository by its owner. Unethical repository collaborators can implant a backdoor into the CI cache before losing their access privileges, we thus measure the number of collaborator removal events as an indicator of this threat. We find that, among the top 5,000 most-starred repositories on GitHub [83], 2,193 (43.86%) of them utilize CI services. Out of these, 994 repositories (45.33%) got a total of 9,900 collaborator removal events, resulting in an average of 10 collaborators being removed per repository. For example, the repository `microsoft/PowerToys` has a total of 108 collaborators removed. The results indicate that collaborator revocation is common and the cache backdoor attack could be a practical security threat.

Git Tag Creation. We conduct a case study by analyzing repositories on GitHub using GitHub Actions. Among 683,125 collected repositories, 91,099 (13.34%) repositories employ the git tag trigger, which triggers the execution of a CI task upon the creation of a git tag. Based on the GHArchive data [60], we find that 69,372 (76.15%) repositories permit at least one collaborator to create tags, indicating that it is very common for a collaborator to create a tag to trigger the execution of a CI task (potentially for launching backdoor attacks).

Repository Transfer. The backdoor attack could also affect repositories that are transferred between different users. We thus evaluate its impact using all collected GitHub repositories from the GHArchive data [60], with their historical owner data included. Among 1,129,876 repositories using CI services on GitHub, we find 42,069 (3.72%) repositories that have been transferred between different users. Particularly, 32 of the top 100 most-starred repositories [83] have been transferred. For example, `vuejs/vue`, the 7th most-starred repository with 203K stars, has been transferred from `yyx990803` (personal account, uid is 499550) to `vuejs` (organizational account, uid is 6128107).

In terms of affected organizations, a total of 29,514 repositories utilize CI services that have been transferred. These repositories belong to 12,360 distinct organizations. Among them, 14,664 were owned by a personal account user, but later transferred to an organization. This type of transfer is common even in large organizations. For example, Google and Microsoft have 1,018 and 559 repositories utilizing CI services on GitHub, respectively. Among them, 60 for Google and 14 for Microsoft were transferred from other users. The results indicate that repository ownership transfer between different users is very common. Thus, once the cache backdoor attack is exploited, it can cause serious damage.

8. Countermeasures and Disclosure

8.1. Best Practices

Check it before save/use it. As cache may be shared among different levels of tasks (e.g., jobs, branches and repositories), developers should carefully configure their cache rules to prevent cache from saving any sensitive data. Definitely, it is always not easy to require developers to set appropriate cache rules to prevent both explicit and implicit cache leaks.

CIPs can adopt CI `secrets` checking mechanisms to find and erase `secrets` in cache, similar to the secure variables masking approach used in build logs [12]. However, replacing secrets in the cache is not always a good idea, as it may disrupt the CI task. Instead, we can integrate *CAnalyzer* into a CI plugin or git plugin for detecting potential secret leakage before the CI caching, thus serving as a timely warning for developers. In addition, the use of in-toto/SLSA attestations that track cache state can prevent cache files from tampering (i.e., cache poisoning attack) [84], [85].

Use explicit cache isolation. CIPs should adopt better default cache isolation strategies. For CIPs like TeamCity and Jenkins that apply a runner host-level isolation strategy, cached files are shared on the same host by default. Developers may be unaware of their implicit cache sharing mechanisms and leak private data accidentally. CIPs can provide cache isolation-related control keywords and ask developers to explicitly define their cache sharing range. Also, disabling implicit cache sharing mechanism can better protect developers from unknowingly leaking sensitive data.

Refine cache permission design. The token T_{cache} should follow the principle of least privilege. The T_{cache} should be only limited to the cache of the specific branch of the originated repository. If CIPs choose the WORM model [18], they should carefully grant one-time write permissions to T_{cache}. In addition, CIPs should optimize authorization with third-party cloud storage services based on the provided security features. For example, AWS S3 provides the *Content-MD5* function [86] and *S3 Object Lock* [18], which can be used to prevent cache objects from being modified.

8.2. Responsible Disclosure

We have promptly communicated with the impacted CIPs to report their vulnerabilities. The CircleCI team have confirmed the cache leakage attack and fixed it immediately. Besides, they also confirmed the cache privilege escalation attack and cross-branch cache poisoning attack. The GitHub team have acknowledged the risk of cache backdoor attack and claimed that they will make the repository transfer functionality more strict in the future. The GitLab team have acknowledged the problem of the cache leakage attack and stated that they plan to refine the best practice proposal. Moreover, the TeamCity and Jenkins teams both have confirmed the cache poisoning attack issue. They recommend using ephemeral CI runners instead of continuous runners. Furthermore, the Bitbucket and TravisCI teams are still reviewing reported issues and considering mitigation.

For 78 vulnerable repositories that potentially leak secrets, we have also tried our best to contact and inform their owners. Particularly, we found the contact information of 47 (60.26%) repositories from GitHub, software website, and several social media platforms, including 42 emails and 5 social media accounts (i.e., Twitter). We have notified these repository owners by sending emails or private messages one by one. So far, we have received a total of 40 replies, of which 22 have been fixed. For example, `swatinem/rust-cache`, a popular Rust cache plugin, has confirmed and fixed its vulnerable cache rules.

9. Related Work

CI Security. Extensive studies have focused on optimizing the build process of CI services [87], [88], [89], [90], [91] and improper CI configurations [10], [92], [93], [94], [95], which may allow attackers to access unauthorized resources and bypass security checks. Koishybayev et al. [4] found that 99.8% of workflows in popular CIPs are over-privileged and developers may pass secrets in plaintext, posing a risk of secret leakage. Li et al. [96] conducted a large-scale measurement on GitHub and revealed lots of CI jobs were abused for illicit cryptomining. Moreover, Gu et al. [8] studied the authentication/authorization process of CI and unveiled multiple security issues related to tokens. Our work is closely related to [8], as we adopt a similar threat model. However, we focus on the CI cache mechanism (which is ignored by previous works) and have identified several new cache-related security issues.

Cache Poisoning. Cache poisoning attacks have been well studied in Web and DNS. For example, Jia et al. [97] proposed a browser cache poisoning attack which enables attackers to launch a Man-in-the-Middle (MitM) attack on HTTPS sessions. There are many web cache poisoning/deception attacks on the web ecosystem, particularly the middle-boxes (e.g., CDN and proxy) [98], [99], [100]. Klein [101] and Kaminsky [102] introduced a series of sophisticated cache poisoning attacks against DNS resolvers and DNS servers. After that, several new DNS cache poisoning attacks against resolvers were proposed, with different assumptions and attack requirements [103], [104], [105], [106], [107]. Different from previous works targeting Web or DNS cache, our work is the first systematic study on CI cache security.

Secret Leakage. Many previous works have studied secret leakage in software developments [108], [109], [110], [111], [112], such as leaked secrets detection in source code [108], [113], [114], [115]. Saha et al. [115] designed a general framework to detect secrets in GitHub repositories. Meli et al. [113] performed a large-scale analysis of API keys embedded directly in source code. Feng et al. [114] showed that a huge number of passwords are leaked in public GitHub repositories. Our work further complements these previous research efforts on studying secret leakage caused by CI caching mechanism flaws. Particularly, even if developers passed their secrets using secret variables to CI tasks, these secrets will still be leaked due to CI caching flaws.

10. Conclusion

This paper systematically analyzes the design and implementation of CI cache and unveils several potential security threats. We find that weak cache isolation and improper cache inheritance commonly exist in mainstream CIPs. We uncover four cache-related attack vectors that allow attackers to inject malicious code or steal sensitive data. To understand the potential impact in the real world, we develop an analysis tool and conduct a large-scale measurement on the open-source community. The results show that many popular repositories and large organizations are potentially affected by these attacks. Also, many repositories expose high-value secrets in the CI cache. We have proposed potential mitigation, duly disclosed the vulnerabilities to related stakeholders, and received positive responses.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. This work was partially supported by the Taishan Scholars Program. The authors from the University of Delaware were partially supported by the National Science Foundation (NSF) grants CNS-2054657 and CNS-2317830.

References

[1] Continuous integration (ci) statistics, trends and facts 2023. https://abdalslam.com/continuous-integration-ci-statistics.

[2] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016.

[3] Caching dependencies - CircleCI. https://circleci.com/docs/caching/.

[4] Igibek Koishybayev, Aleksandr Nahapetyan, Raima Zachariah, Siddharth Muralee, Bradley Reaves, Alexandros Kapravelos, and Aravind Machiry. Characterizing the security of github CI workflows. In *31st USENIX Security Symposium (USENIX Security)*, 2022.

[5] Hackers backdoor php source code after breaching internal git server. https://arstechnica.com/gadgets/2021/03/hackers-backdoor-php-source-code-after-breaching-internal-git-server/.

[6] Datadog rotates RPM signing key exposed in CircleCI hack. https://www.bleepingcomputer.com/news/security/datadog-rotates-rpm-signing-key-exposed-in-circleci-hack/.

[7] CircleCI incident report for january 4, 2023 security incident. https://circleci.com/blog/jan-4-2023-incident-report/.

[8] Yacong Gu, Lingyun Ying, Huajun Chai, Chu Qiao, Haixin Duan, and Xing Gao. Continuous Intrusion: Characterizing the Security of Continuous Integration Services. In *Proceedings of the 44th IEEE Symposium on Security and Privacy*, 2023.

[9] P. Ladisa, H. Plate, M. Martinez, and O. Barais. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2023.

[10] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. An Empirical Characterization of Bad Practices in Continuous Integration. *Empirical Software Engineering*, 2020.

[11] Encrypted secrets - github docs. https://docs.github.com/en/actions/security-guides/encrypted-secrets.

[12] Atlassian. Bitbucket cloud variables and secrets. https://support.atlassian.com/bitbucket-cloud/docs/variables-and-secrets/.

[13] Assigning permissions to jobs. https://docs.github.com/en/actions/using-jobs/assigning-permissions-to-jobs.

[14] Job permissions - permissions and roles. https://docs.gitlab.com/ee/user/permissions.html#job-permissions.

[15] Caching dependencies to speed up workflows. https://docs.github.com/en/actions/using-workflows/caching-dependencies-to-speed-up-workflows.

[16] Pull request builds and caches-caching dependencies and directories. https://docs.travis-ci.com/user/caching/#pull-request-builds-and-caches.

[17] Mitmproxy Project. mitmproxy - an interactive https proxy. https://mitmproxy.org/.

[18] Using s3 object lock - amazon simple storage service. https://docs.aws.amazon.com/AmazonS3/latest/userguide/object-lock.html.

[19] Generating a presigned url to upload an object - amazon simple storage service. https://docs.aws.amazon.com/AmazonS3/latest/userguide/PresignedUrlUploadObject.html.

[20] Sharing objects using presigned urls. https://docs.aws.amazon.com/AmazonS3/latest/userguide/ShareObjectPreSignedURL.html.

[21] Using temporary credentials with AWS resources - AWS identity and access management. https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_temp_use-resources.html,.

[22] Job cacher. https://plugins.jenkins.io/jobcacher.

[23] Jenkins community announces record growth and innovation in 2017. https://www.cloudbees.com/newsroom/jenkins-community-announces-record-growth-and-innovation-2017.

[24] Use the same cache for all branches. https://docs.gitlab.com/ee/ci/caching/#use-the-same-cache-for-all-branches.

[25] Merge request pipelines. https://docs.gitlab.com/ee/ci/pipelines/merge_request_pipelines.html#use-with-forked-projects.

[26] Build open source projects. https://circleci.com/docs/oss/.

[27] Github doc - about custom actions. https://docs.github.com/en/actions/creating-actions/about-custom-actions.

[28] Circleci doc - orbs overview. https://circleci.com/docs/orb-intro/.

[29] actions/cache. https://github.com/actions/cache.

[30] Swatinem/rust-cache. https://github.com/Swatinem/rust-cache.

[31] docker/login-action. https://github.com/docker/login-action.

[32] About protected branches. https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/managing-protected-branches/about-protected-branches.

[33] Permissions and roles, gitlab docs. https://docs.gitlab.com/ee/user/permissions.html.

[34] Configuring tag protection rules, github docs. https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/managing-repository-settings/configuring-tag-protection-rules.

[35] Protected tags, gitlab docs. https://docs.gitlab.com/ee/user/project/protected_tags.html.

[36] Keeping your github actions and workflows secure - github security lab. https://securitylab.github.com/research/github-actions-preventing-pwn-requests/.

[37] Merge request pipelines – gitlab ci/cd. https://docs.gitlab.com/ee/ci/pipelines/merge_request_pipelines.html.

[38] package-locks - npm Docs. https://docs.npmjs.com/cli/v6/configuring-npm/package-locks.

[39] Requirements File Format - pip Docs. https://pip.pypa.io/en/stable/reference/requirements-file-format/.

[40] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. Trade-offs in continuous integration: Assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[41] Open Zipkin. openzipkin/zipkin-dependencies. https://github.com/openzipkin/zipkin-dependencies.

[42] openzipkin/zipkin-dependencies docker image. https://hub.docker.com/r/openzipkin/zipkin-dependencies.

[43] docker login. https://docs.docker.com/engine/reference/commandline/login/.

[44] Liquibase Inc. Liquibase. https://github.com/liquibase/liquibase.

[45] Gitlab ci templates/gradle.gitlab-ci.yml. https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Gradle.gitlab-ci.yml.

[46] Gradle build environment. https://docs.gradle.org/current/userguide/build_environment.html#sec:gradle_configuration_properties.

[47] Maven - caching strategies - CircleCI. https://circleci.com/docs/caching-strategy/#maven-java-and-leiningen-clojure.

[48] Maven – settings reference. https://maven.apache.org/settings.html.

[49] Gradle - caching strategies - CircleCI. https://circleci.com/docs/caching-strategy/#gradle-java.

[50] Arbitrary directories - caching dependencies and directories - TravisCI. https://docs.travis-ci.com/user/caching/#arbitrary-directories.

[51] Rust cargo cache - caching dependencies and directories - TravisCI. https://docs.travis-ci.com/user/caching/#rust-cargo-cache.

[52] cargo login - the cargo book. https://doc.rust-lang.org/cargo/commands/cargo-login.html.

[53] jsonc-parser package - cargo. https://crates.io/crates/jsonc-parser.

[54] Sudo CircleCI Make Me a Sandwich. https://circleci.com/blog/sudo-circleci-make-me-a-sandwich/.

[55] Restrictions for accessing a cache - github docs. https://docs.github.com/en/actions/using-workflows/caching-dependencies-to-speed-up-workflows#restrictions-for-accessing-a-cache.

[56] Projects API - GitLab. https://docs.gitlab.com/ee/api/projects.html.

[57] List Public Repositories - The Bitbucket Cloud REST API. https://developer.atlassian.com/cloud/bitbucket/rest/api-group-repositories/.

[58] Repository Files API - GitLab. https://docs.gitlab.com/ee/api/repository_files.html.

[59] Get File or Directory Contents - The Bitbucket Cloud REST API. https://developer.atlassian.com/cloud/bitbucket/rest/api-group-source/#api-repositories-workspace-repo-slug-src-post.

[60] GH Archive. https://www.gharchive.org/.

[61] Github data, ready for you to explore with bigquery. https://github.blog/2017-01-19-github-data-ready-for-you-to-explore-with-bigquery/.

[62] Protected Branches. https://docs.github.com/en/rest/branches/branch-protection.

[63] Changing Your GitHub Username. https://docs.github.com/en/account-and-profile/setting-up-and-managing-your-github-user-account/managing-user-account-settings/changing-your-github-username.

[64] List repository collaborators - github api. https://docs.github.com/en/rest/collaborators/collaborators?apiVersion=2022-11-28#list-repository-collaborators.

[65] List pull requests - github api. https://docs.github.com/en/rest/pulls/pulls?apiVersion=2022-11-28#list-pull-requests.

[66] YAML ain't markup language (YAML™) revision 1.2.2. https://yaml.org/spec/1.2.2/.

[67] Vitaly Chaykovsky. strace: linux syscall tracer. https://strace.io/.

[68] trufflesecurity. Trufflehog: Find and verify credentials. https://github.com/trufflesecurity/trufflehog.

[69] Kernel probes (kprobes). https://docs.kernel.org/trace/kprobes.html.

[70] Patterns to match file paths - workflow syntax for github actions. https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions#patterns-to-match-file-paths.

[71] Cache paths. https://docs.gitlab.com/ee/ci/yaml/#cachepaths.

[72] Configuration reference. https://circleci.com/docs/configuration-reference/#savecache.

[73] Use glob patterns on the pipelines yaml file. https://support.atlassian.com/bitbucket-cloud/docs/use-glob-patterns-on-the-pipelines-yaml-file/.

[74] Build phases - caching dependencies and directories – travisci. https://docs.travis-ci.com/user/caching/#build-phases.

[75] gluon-lang/gluon. https://github.com/dprint/gluon-lang/gluon.

[76] dprint-plugin-typescript. https://github.com/dprint/dprint-plugin-typescript/blob/f2686fd596b75a945b869263336fbf9316be9dcc/.github/workflows/ci.yml.

[77] pydatalog. https://sites.google.com/site/pydatalog/home.

[78] Vulnerability disclosure faq. https://googleprojectzero.blogspot.com/p/vulnerability-disclosure-faq.html.

[79] List tag protection states for a repository, github rest api docs. https://docs.github.com/en/rest/repos/tags?apiVersion=2022-11-28#list-tag-protection-states-for-a-repository.

[80] The liquibase community. https://www.liquibase.org/.

[81] chalharu/rust-nearly-eq. https://github.com/chalharu/rust-nearly-eq.

[82] List pull requests - github restful api. https://docs.github.com/en/rest/pulls/pulls?apiVersion=2022-11-28#list-pull-requests.

[83] Gitstar ranking. https://gitstar-ranking.com/.

[84] in-toto, a framework to secure the integrity of software supply chains. https://in-toto.io/.

[85] Slsa specification. https://slsa.dev/spec/v1.0/.

[86] Generate a pre-signed url for an amazon s3 put operation with a specific payload. https://docs.aws.amazon.com/sdk-for-go/v1/developer-guide/s3-example-presigned-urls.html#generate-a-pre-signed-url-put.

[87] Bihuan Chen, Linlin Chen, Chen Zhang, and Xin Peng. Buildfast: History-aware build outcome prediction for fast feedback and reduced cost in continuous integration. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2021.

[88] Tingting Yu and Ting Wang. A study of regression test selection in continuous integration environments. In *IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, 2018.

[89] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 2017.

[90] Dominik Wermke, Noah Wöhler, Jan H. Klemmer, Marcel Fourné, Yasemin Acar, and Sascha Fahl. Committed to trust: A qualitative study on security trust in open source software projects. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[91] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. The impact of continuous integration on other software development practices: A large-scale empirical study. In *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[92] Keheliya Gallaba and Shane McIntosh. Use and misuse of continuous integration features: An empirical study of projects that (mis)use travis ci. *IEEE Transactions on Software Engineering*, 2020.

[93] Wagner Felidré, Leonardo Furtado, Daniel A. da Costa, Bruno Cartaxo, and Gustavo Pinto. Continuous integration theater. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019.

[94] Akond Rahman, Chris Parnin, and Laurie Williams. The seven sins: Security smells in infrastructure as code scripts. In *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

[95] Carmine Vassallo, Sebastian Proksch, Harald C. Gall, and Massimiliano Di Penta. Automated reporting of anti-patterns and decay in continuous integration. In *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

[96] Zhi Li, Weijie Liu, Hongbo Chen, XiaoFeng Wang, Xiaojing Liao, Luyi Xing, Mingming Zha, Hai Jin, and Deqing Zou. Robbery on devops: Understanding and mitigating illicit cryptomining on continuous integration service platforms. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[97] Yaoqi Jia, Yue Chen, Xinshu Dong, Prateek Saxena, Jian Mao, and Zhenkai Liang. Man-in-the-browser-cache: Persisting https attacks via browser cache poisoning. *Computers Security*, 2015.

[98] James Kettle. Practical web cache poisoning: Redefining unexploitable. https://i.blackhat.com/us-18/Thu-August-9/us-18-Kettle-Practical-Web-Cache-Poisoning-Redefining-Unexploitable.pdf.

[99] Hoai Viet Nguyen, Luigi Lo Iacono, and Hannes Federrath. Your cache has fallen: Cache-poisoned denial-of-service attack. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[100] Seyed Ali Mirheidari, Sajjad Arshad, Kaan Onarlioglu, Bruno Crispo, Engin Kirda, and William Robertson. Cached and confused: Web cache deception in the wild. In *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020.

[101] Amit Klein. Bind 8 dns cache poisoning. https://dl.packetstormsecurity.net/papers/attack/BIND_8_DNS_Cache_Poisoning.pdf.

[102] Dan Kaminsky. Black ops 2008: It's the end of the cache as we know it. https://www.blackhat.com/presentations/bh-jp-08/bh-jp-08-Kaminsky/BlackHat-Japan-08-Kaminsky-DNS08-BlackOps.pdf.

[103] Amir Herzberg and Haya Shulman. Security of patched dns. In *Computer Security – ESORICS 2012*, 2012.

[104] Amir Herzberg and Haya Shulman. Vulnerable delegation of dns resolution. In *Computer Security – ESORICS 2013*, 2013.

[105] Haya Shulman and Michael Waidner. Fragmentation considered leaking: Port inference for dns poisoning. In *Applied Cryptography and Network Security*, 2014.

[106] Keyu Man, Zhiyun Qian, Zhongjie Wang, Xiaofeng Zheng, Youjun Huang, and Haixin Duan. Dns cache poisoning attack reloaded: Revolutions with side channels. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[107] Keyu Man, Xin'an Zhou, and Zhiyun Qian. Dns cache poisoning attack: Resurrections with side channels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[108] Vibha Singhal Sinha, Diptikalyan Saha, Pankaj Dhoolia, Rohan Padhye, and Senthil Mani. Detecting and mitigating secret-key leaks in source code repositories. In *IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015.

[109] Konstantinos Chatzikokolakis, Tom Chothia, and Apratim Guha. Statistical measurement of information leakage. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2010.

[110] Min Xu, Armin Namavari, David Cash, and Thomas Ristenpart. Searching encrypted data with size-locked indexes. In *USENIX Security Symposium*, 2021.

[111] Michael Backes, Boris Köpf, and Andrey Rybalchenko. Automatic discovery and quantification of information leaks. In *30th IEEE Symposium on Security and Privacy (S&P)*, 2009.

[112] AD Diego. *Automatic extraction of API Keys from Android applications*. PhD thesis, Ph. D. dissertation, UNIVERSITA DEGLI STUDI DI ROMA" TOR VERGATA, 2017.

[113] Michael Meli, Matthew R McNiece, and Bradley Reaves. How bad can it git? characterizing secret leakage in public github repositories. In *NDSS*, 2019.

[114] Runhan Feng, Ziyang Yan, Shiyan Peng, and Yuanyuan Zhang. Automated detection of password leakage from public github repositories. In *Proceedings of the 44th International Conference on Software Engineering*, 2022.

[115] Aakanksha Saha, Tamara Denning, Vivek Srikumar, and Sneha Kumar Kasera. Secrets in source code: Reducing false positives using machine learning. In *International Conference on COMmunication Systems NETworkS (COMSNETS)*, 2020.

Appendix A.

TABLE 7: The comprehensive list of shell tools and plugins used in CI tasks that create sensitive files, among the top 100 most-used shell tools as well as the top 100 most-used plugins for GitHub Actions and CircleCI. * denotes shell tools, † denotes GitHub plugins, and ‡ denotes CricleCI plugins.

#	Type	Tools / Plugins	Commands / Parameters	Generated Sensitive Files	Sensitive Data
1	*	docker	$ docker login	~/.docker/config.json	Docker registry login credentials.
			$ docker trust key load	~/.docker/trust/private/*.key	Private key for signing/verifying Docker images.
2	*	npm	$ npm login	~/.npmrc	npm registry login credentials.
3	*	aws	$ aws configure set aws_secret_access_key	~/.aws/credentials	AWS services access key secret.
			$ aws configure set aws_access_key_id	~/.aws/credentials	AWS services access key id.
4	*	npx	$ npx vercel −−token	~/.npm/_logs/*.log	Vercel services access credentials.
5	*	poetry	$ poetry config http-basic.pypi __token__	~/.config/pypoetry/auth.toml	PyPI registry login credentials.
6	*	cargo	$ cargo login	~/.cargo/credentials	API credentials for publishing Cargo packages.
7	*	yarn	$ yarn config set npmAuth-Token	/usr/local/share/.yarnrc	API credentials for publishing npm packages.
8	*	az	$ az login	~/.azure/msal_token_cache.json	Azure Cloud services access credentials.
9	*	b2	$ b2 authorize-account	~/.b2_account_info	Backblaze services access credentials.
10	*	gcloud	$ gcloud auth login	~/.config/gcloud/logs/*.log ~/.config/gcloud/legacy_credentials/$account/.boto ~/.config/gcloud/configurations/config_default ~/.config/gcloud/legacy_credentials/$account/adc.json	Google Cloud services access credentials.
11	*	nuget	$ nuget setapikey	~/.config/NuGet/NuGet.Config	NuGet login credentials.
12	*	oc	$ oc login	~/.kube/config	Container services access credentials.
13	*	helm	$ helm registry login	~/.config/helm/registry/config.json	Container registry login credentials.
14	*	composer	$ composer config −−global −−auth	~/.config/composer/auth.json	API credentials for publishing Composer packages.
15	*	vercel	$ vercel build −−token	.vercel/output/builds.json	Vercel services access credentials.
16	*	heroku	$ heroku login	~/.netrc	Heroku services access credentials.
17	*	anaconda	$ anaconda login	~/.config/binstar/$host.token	Anaconda services access credentials.
18	*	sfdx	$ sfdx force:auth:device:login $ sfdx force:auth:jwt:grant	~/.sfdx/$account.json ~/.sfdx/$account.json	SalesForce services access credentials.
19	*	vagrant	$ vagrant cloud auth login	~/.vagrant.d/data/vagrant_login_token	Vagrant services access credentials.
20	*	ibmcloud	$ ibmcloud login	~/.bluemix/config.json	IBM Cloud services access credentials.
21	*	cosign	$ cosign login	~/.docker/config.json	Docker registry login credentials.
22	*	skopeo	$ skopeo login	/run/user/0/containers/auth.json	Container registry login credentials.
23	*	cf	$ cf login	~/.cf/config.json	Container registry login credentials.
24	*	snyk	$ snyk auth	~/.config/configstore/snyk.json	Snyk services access credentials.
25	*	doctl	$ doctl registry login	~/.docker/config.json	DigitalOcean registry access credentials.
26	*	podman	$ podman login	/run/user/0/containers/auth.json	Podman services access credentials.
27	†	docker/login-action	password	~/.docker/config.json	Docker registry login credentials.
28	†	FirebaseExtended/action-hosting-deploy	firebaseServiceAccount	/tmp/tmp-*.json	Firebase services access credentials.
29	†	crazy-max/ghaction-import-gpg	gpg_private_key	~/.gnupg/private-keys-v1.d/*.key	GPG private key.
30	†	akhileshns/heroku-deploy	heroku_api_key	~/.netrc	Heroku services access credentials.
31	†	hashicorp/setup-terraform	cli_config_credentials_token	~/.terraformrc	Terraform services access credentials.
32	†	azure/docker-login	password	/home/runner/work/_temp/ docker_login_$timestamp/config.json	Docker registry login credentials.
33	†	r0adkll/sign-android-release	releaseDirectory, signingKeyBase64	./$releaseDirectory/signingKey.jks	Android app signing key.
34	†	shimataro/ssh-key-action	key	~/.ssh/id_rsa	SSH Key.
35	†	whelk-io/maven-settings-xml-action	servers	~/.m2/settings.xml	Maven registry access credentials.
36	‡	circleci/aws-cli	aws-secret-access-key	~/.aws/credentials	AWS services access credentials.
37	‡	circleci/aws-ecr	aws-secret-access-key	~/.aws/credentials	AWS services access credentials.
			dockerhub-password	~/.docker/config.json	Docker registry login credentials.
38	‡	circleci/aws-s3	aws-secret-access-key	~/.aws/credentials ~/.aws/config	AWS services access credentials.
39	‡	circleci/aws-ecs	aws-secret-access-key	~/.aws/credentials ~/.aws/config	AWS services access credentials.
40	‡	circleci/docker	docker-password	~/.docker/config.json	Docker registry login credentials.
41	‡	circleci/android	base64-keystore google-play-key	./keystore ./keystore.properties ./google-play-key.json	Google Play services access credentials.
42	‡	circleci/kubernetes	kubeconfig	~/.kube/config	Kubernetes services access credentials
43	‡	circleci/gcp-cli	gcloud_service_key	~/gcp_cred_config.json ~/gcloud-service-key.json ~/.config/gcloud/$account/adc.json	Google Cloud services access credentials.
44	‡	circleci/heroku	api-key	/tmp/orblog.txt	Heroku services access credentials.
45	‡	circleci/gcp-gcr	gcloud_service_key	~/gcloud-service-key.json ~/.config/gcloud/legacy_credentials/$account/adc.json	Google Cloud services access credentials.
46	‡	circleci/gcp-gke	gcloud_service_key	~/gcloud-service-key.json ~/.config/gcloud/legacy_credentials/$account/adc.json	Google Cloud services access credentials.
47	‡	circleci/gcp-cloud-run	gcloud_service_key	~/gcloud-service-key.json ~/.config/gcloud/legacy_credentials/$account/adc.json	Google Cloud services access credentials.
48	‡	circleci/aws-code-deploy	aws-secret-access-key	~/.aws/credentials ~/.aws/config	AWS services access credentials.

Appendix B. Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

This work analyzes cache implementations of seven popular continuous integration services to identify vulnerabilities related to unauthorized access/modification of sensitive cache objects. The authors find that different CI services use different rules for sharing cache objects across jobs/branches/repos, which can lead to attacks when combined with poorly-written CI jobs. The paper lists different attack goals (improper write-up, improper read-up, improper token permission, improper cache eviction). It describes which cache directives (and platform defaults) can be used to achieve these goals. The authors built the Canalyzer tool to scan public repositories with CI jobs and cache directives to address the vulnerabilities.

B.2. Scientific Contributions

- Independent Confirmation of Important Results with Limited Prior Research.
- Identifies an Impactful Vulnerability.
- Creates a New Tool to Enable Future Science.
- Provides a Valuable Step Forward in an Established Field.

B.3. Reasons for Acceptance

1) The paper illustrates a novel cache-based attack on CI/CD systems and demonstrates that open-source projects are vulnerable to those attacks.
2) The authors present the CAnalyzer tool that can be used to scan public repositories for the vulnerabilities illustrated in the paper.