





SoK: Understanding CI/CD Security: A Comprehensive Review of Architecture, Attacks, and Defenses

1st Ryan Zmuda 
Secure and Resilient Systems
Riverside Research
Beavercreek, OH, USA
rzmuda@riversideresearch.org

2nd Russell Graves 
Secure and Resilient Systems
Riverside Research
Beavercreek, OH, USA
rgraves@riversideresearch.org

3rd Michael Shepherd 
Secure and Resilient Systems
Riverside Research
Lexington, MA, USA
mshepherd@riversideresearch.org

4th Scott Brookes 
Secure and Resilient Systems
Riverside Research
Lexington, MA, USA
sbrookes@riversideresearch.org

Abstract—Maintaining the confidentiality, availability, and integrity of CI/CD systems against attacks which target their pipelines poses a significant security challenge. Attacks like Solarwinds’s Orion Sunburst and CodeCov’s Secret Exfiltration demonstrate that the abuse of simple misconfigurations spanning the vastly complex stack of services which make up CI/CD pipelines can have devastating consequences. In this paper, we quantify and consolidate CI/CD terminology, present a system architecture characterizing CI/CD, analyze patterns in existing CI/CD attacks, categorize popular free and open source defense tools, and perform gap analysis evaluation on existing tooling. We find that state-of-the-art CI/CD defense tools fail to mitigate 5 out of the 7 evaluation attacks we created.

I. INTRODUCTION

The Continuous Integration and Continuous Delivery / Deployment (CI/CD) paradigm has garnered an immense level of use and support over the last decade, shaping and redefining almost every aspect of the software delivery process. At its core, CI/CD uses automated pipelines for testing, building, and deploying software. The term DevOps was coined from the increased collaboration between both Development and Operations teams. The newer Development-Security-Operations (DevSecOps) focus has emerged from DevOps in response to an increased industry focus on security.

The difficulty in securing CI/CD pipelines stems from the increased level of abstraction and centralization of previously disjoint parts of the build and release software development lifecycle. This shift in environment has resulted in a concentration of desirable targets, like secret keys and credentials, into one convenient target for attackers. The collation of these desirable targets expands the attacker’s capacity for lateral movement upon breach by dissolving otherwise traditional segmentation between services.

While the main characterizing feature of CI/CD has been automated pipelines, the entirety of the build and deployment

chain is reliant on a trusted platform provider (like a GitLab instance), the broader software supply chain (npm, docker hub, etc), and the security of the organization’s infrastructure that executes the pipeline. This composition of services complicates quantifying the full extent of the Trusted Compute Base (TCB) of a CI/CD environment.

CI/CD pipelines must be able to access private source code and APIs, various internal and external package registries, external repositories for artifact compilation or product orchestration, and other capabilities as needed in order to fulfill its role as a development and deployment multi-tool. This extensive set of capabilities and permissions necessitates security validation. Attacks like SolarWinds Sunburst [1] demonstrate the reach and impact of compromising an automated build and deployment pipeline, in which a malicious version of the Orion software propagated from an infected build server downstream to affect more than 18,000 business and government customers.

A CI/CD pipeline’s unique reliance on novel and established supply chain sources poses a significant challenge for securing the pipeline against software supply chain vulnerabilities. In the open source community, the reusability of Pipelines-as-Packages (PaPs) has expanded the reach of the supply chain directly into the pipeline stages themselves. Reusable pipeline stages can lead to a cascade of vulnerabilities from just one compromised upstream dependency. An example of such an attack would be the breach in CodeCov [2], where an infected CodeCov release propagated into dependent pipelines, exfiltrating their secrets to a centralized server.

Investigating and improving measures specific to the security of CI/CD pipelines is paramount to addressing the growing complexity and intricacy of CI/CD systems and their attacks. We offer the following research questions to help guide the efficient addressing of future CI/CD security challenges:

- RQ1. What components typically make up the trusted execution environment of CI/CD pipelines, and how are they arranged?
- RQ2. What classes of attacks are used against CI/CD pipelines, and what are their outcomes?
- RQ3. What approaches or tools exist to protect the integrity, validity, or availability of CI/CD pipelines?
- RQ4. How do existing CI/CD security tools perform against established vulnerability patterns?

We answer these questions by exploring and formalizing CI/CD system architecture and attack surfaces in Section IV, aggregating and categorizing CI/CD attacks in Section V, analyzing and classifying tools for securing CI/CD pipelines in Section VI, and performing novel CI/CD security gap analysis in Section VII.

II. BACKGROUND

The DevOps methodology arose between 2007 and 2008 due to a cultural shift towards an increase in automation and collaboration between development, operations, and quality assurance teams. In more recent years, an increased attention on introducing security practices into DevOps has led to the DevSecOps methodology that advocates for a shift left in security practices, to earlier into the development cycle. Both DevOps and DevSecOps rely on CI/CD, a pattern comprised of continuous integration, continuous delivery, and continuous deployment practices.

- Continuous integration (CI) involves frequently merging code changes into a central repository, which is typically equipped with automated testing. The iterative process of frequently integrating smaller patches allows each change to be thoroughly tested in isolation, identifying bugs much earlier on in the development cycle.
- Continuous delivery (CD) automates the deployment process of an artifact or service, up to a final step for manual approval. This ensures that the target software is always ready to be deployed to production, while allowing manual review and approval.
- Continuous deployment (CD) is similar to continuous delivery, but there is no manual intervention involved. All of the validation is built into the pipeline, and any resulting artifact is automatically approved contingent on its automated tests passing.

Within continuous delivery and continuous deployment, there are multiple models of deployment into production environments. Push-based deployments require a deployment pipeline to package and export a final artifact to an external service, while pull-based deployments utilize external deployment agents configured to continually query a source for new configurations or artifacts to deploy.

Most CI/CD pipelines are linked in some way to a Source Code Management (SCM) solution, which in turn is based on a Version Control System (VCS) like git or subversion. Execution of the pipeline is typically triggered automatically based on events in the SCM solution’s codebase, like pull requests or

git commits. Pipelines are often defined within a SCM or VCS repository itself as declarative configuration files. Defining platform level services and infrastructure through declarative configurations is commonly referred to as Infrastructure-as-Code (IaC).

The architecture of CI/CD environments is complex and varies greatly based on the platform or repository. Despite these differences, each pipeline is universally composed of stages that are sent to an executor or runner. A stage (or job) is a sequence of actions, which can be combined to form a greater pipeline. Some platforms refer to the combination of jobs as “workflows”, but they are functionally the same as the pipeline itself.

Pipelines are interpreted by executors, which sequentially perform operations specified by the pipeline in its IaC. Executors can be standalone, but they are typically implemented inside a runner. Runners are individual on-demand agents that accept pipeline dispatches. These runners are usually virtual machines or containers executing on a server hosted in the cloud by the SCM provider or self-hosted by the repository maintainers behind a firewall.

Many SCM provider companies offer CI/CD pipeline solutions that are either built-in to the platform service, like GitHub Actions or GitLab Pipelines, or act as a standalone entity, like Jenkins. Provider solutions can be utilized through managed cloud instances or deployed in self-hosted environments, as illustrated in Table I. This distinction clarifies the vulnerability space of the pipeline execution environment, especially concerning self-hosted environments that require additional maintenance and security efforts.

TABLE I
Common CI/CD Providers and Their Environments.
Self-hosted environments require additional security considerations.

Provider	Cloud	Self Hosted
AWS CodePipeline	Yes	No
Google Cloud Pipeline	Yes	No
IBM Cloud CD	Yes	No
Oracle Cloud DevOps	Yes	No
GitHub Actions	Yes	Yes
GitLab Pipelines	Yes	Yes
BitBucket Pipelines	Yes	Yes
CircleCI	Yes	Yes
TravisCI	Yes	Yes
Azure Pipelines	Yes	Yes
Jetbrains TeamCity	Yes	Yes
Jenkins	No	Yes

Pipeline runners from cloud providers are typically available either for free with usage quotas, or on a paid enterprise plan. Many of these providers allow customers to supplement their provided cloud runners with separate, self-hosted runners. This convention is generally discouraged as self-hosted runners have a higher capacity for abuse [3]. Some providers, like TravisCI and CircleCI, integrate directly into other providers SCM solution through hooks. This allows developers to use GitHub as a SCM platform, but execute their pipelines on an external TravisCI or CircleCI server.

III. RELATED RESEARCH

Existing CI/CD security research has focused primarily around the GitHub Actions ecosystem, studying both the propagation of upstream vulnerabilities from reusable actions [4], [5], and the lack of proper security configuration practices on many GitHub repositories [6], [7]. There is, however, comparatively less research in the exploration and analysis of other CI/CD platforms.

CI/CD vulnerability mitigation research exists, but not in a comprehensive or cross-domain manner. Best practices have been examined for the integration of state-of-the-art (SOTA) defense tooling [8], including the presentation of methodologies like VeriDevOps [9], which provides a set of recommendations for developers to maintain security within their DevOps environments. Others discuss the integration of threat modeling into pipelines to improve vulnerability discovery [10], [11], and the importance of test automation within CI/CD [12]. Experiments have been performed with integrating fuzzing stages into CI/CD pipelines, mainly focused on the potential viability of short term fuzzing [13], [14]. These experiments could lay the foundation for more advanced continuous testing tools.

Many researchers have examined CI/CD supply chain security through the usage of knowledge graphs and SBOMs (Software Bill-Of-Materials) [15], [16], formalizing desirable security properties necessary to mitigate supply chain attacks. Additionally, the security of critical CI/CD components, like containers, has been studied extensively, with research introducing many mechanisms to understand and audit their contents or behavior [17]–[19].

Many disjoint pieces of the CI/CD security puzzle have been explored in academia, but there is no cohesive representation of security specifically for CI/CD pipelines covering all their components across multiple platforms. This paper seeks to provide this aggregation by introducing a novel CI/CD system architecture denoting attack surfaces, a classification framework for existing in-the-wild attacks, an enumeration of CI/CD security tools, and an evaluation of existing security tools against controlled evaluation attacks to better understand the gaps within modern CI/CD tooling.

IV. PIPELINE CHARACTERIZATION

Answering RQ1 requires a complete compositional understanding of the trusted execution environment and software supply chain interface of CI/CD pipelines. Achieving this understanding can be difficult, however, due to the scale and complexity of these systems. As a product of our research, we present Figure 1 as a characterization of a common CI/CD system architecture to assist in the visualization and understanding of CI/CD systems.

A. CI/CD System Architecture

While CI/CD execution environments vary greatly, there are a few common patterns between architectures. A pipeline is triggered by a hook from the SCM or other external sources and is then instantiated within a runner. This pipeline runner

is made available by a provider or service which enables the CI/CD execution interface. Providers of CI/CD services also commonly host the SCM or VCS that generate the hooks to the CI/CD execution interface. This provider is a service running on the platform, which can be self-hosted locally or managed in the cloud, alongside many other services like security solutions, ticketing software, and other programs. Many external services are accessed during a typical pipeline execution, such as library hosts (e.g. npm, pypi), operating system package providers (e.g. apt, nixpkgs), and container registries (e.g. Docker Hub, GitHub Container Registry).

The platform is broadly referred to as the hardware and software layers which enable the services that run the CI/CD pipeline. For example, the platform running a self-hosted GitLab instance would be the physical server, the operating system, and any virtualization layers. The Admin (A) in Figure 1 is the individual responsible for the creation and maintenance of the platform. Services like SCM, CNAPPs, ticketing, and the CI/CD provider itself all run concurrently on the platform.

The DevOps Engineer (E) in Figure 1 is responsible for the construction, configuration, and maintenance of the CI/CD pipeline configuration. This configuration specifies the stages, dependencies, tools, and directions for a pipeline execution. A dispatched runner will proceed through individual stages when instructed to execute as specified in the pipeline’s configuration file. This process is represented as steps within the pipeline section in Figure 1. Common stages like building, testing, and deploying are all performed as subsequent interconnected steps. It is typical for the deployment stages of pipelines to end with pushing an artifact to an external location, such as a package registry or artifactory. For our purposes, we refer to such deployments as being shipped into the production environment.

The Developer (D) in Figure 1 is an outside user of the Version Control or SCM solution, to which the pipeline is connected. The developer will submit patches or changes to a codebase, to which the Provider will automatically fire hooks that spawn pipelines the DevOps Engineer (E) has configured. These pipelines will then operate on the patch content, performing builds, tests, and more. For providers like GitHub, hooks can also be fired due to Pull Request (PR) or Issue events, which allow pipelines to perform more complex tasking like testing PR code or running triaging tools on bug reports.

B. Supply Chain

Pipelines access a variety of supply chain sources, both internally and externally. Library sources like npm, pypi, and cargo provide both packages to the artifact and tools for the pipeline execution itself. Repositories like nixpkgs and the Ubuntu package repository provide operating system packages like compilers or fuzzers to the pipeline. Reusable pipeline components have also become critical to modern CI/CD development, especially on platforms like GitHub. GitHub provides the “GitHub Marketplace,” a service that allows users to publish reusable GitHub Actions or Workflows, enabling

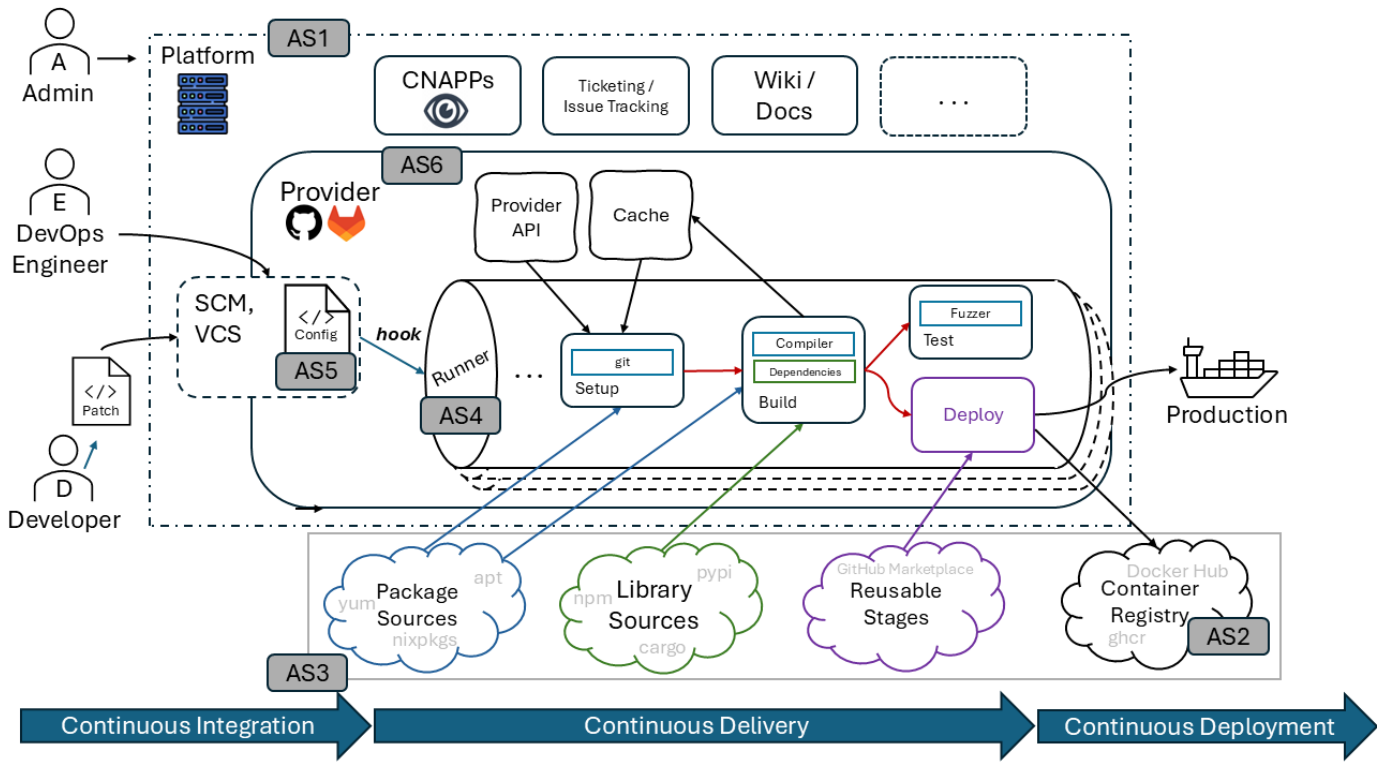


Fig. 1: A graphic illustrating the composition of a CI/CD environment. DevOps Engineers (E) and Admins (A) work together to maintain the CI/CD environment, used by Developers (D). A complex execution process begins with an event hook from an SCM or VCS. Several normally disjoint Attack Surfaces are present in this one system.

generic stages like Docker Buildx Setup [20] to be inserted into thousands of dependent pipelines.

While these reusable stages increase the productivity of workflows, they also increase their supply chain attack surface. The recent tj-actions supply chain attack [21] demonstrates that even reputable and widely used reusable actions can be compromised. Reusable stages themselves are capable of relying on other reusable stages, meaning in practice that one upstream compromise of a reusable action can reach an enormous amount of downstream dependencies. Research has shown that within a set of 447,238 GitHub Actions workflows, 97% of them were using at least one unverified reusable action [4].

In addition to the threat posed by reusable stages, novel CI/CD supply chain attacks like cache poisoning have recently begun to emerge [22]. As pipelines work with a large amount of packages and data, reusable stages like GitHub's Cache Action [23] allow for storing intermediate files and artifacts between pipeline runs. It has been demonstrated that these caches can be trivially poisoned, and often face little to no validation [24], [25]. Recent attacks have leveraged leaked credentials to replace cached objects with malicious ones through platform APIs [22]. Once infected, these cached objects are downloaded by subsequent pipelines, posing a number of security risks.

Containerization also plays a large role in pipeline ex-

ecution. Many providers execute their pipeline runners as containers, like the GitHub Actions Runner [26]. Container Registries like ChainGuard, Docker Hub, and the GitHub Container Registry provide container images used as both bases for derivative images built by the pipeline, and images for the runners which execute the stages. Additionally, it is common practice to run individual stages within a larger pipeline in separate containers, and many providers expose interfaces for this in their pipeline configuration specification [27].

C. Attack Surface

Due to the scope of trusted components which make up a CI/CD pipeline, CI/CD security tools are disjoint and multifaceted. Many tools seek to secure one attack surface of a pipeline, by restricting network traffic [28]–[30], or parsing IaC for errors [31]–[34]. We identify the following 6 vulnerable attack surfaces within CI/CD pipelines:

- **AS1) The Platform:**

Server and operating system-level security targets, including hosted services such as ticketing, issue tracking, documentation, and wiki software; CI/CD platform security is largely an extension of existing platform security.

- **AS2) Containers:**

Virtualized execution environments commonly used within a pipeline; contain a breadth of potentially vulner-

TABLE II

An aggregation of well documented CI/CD related attacks, classified with their exploited attack surface and outcomes. The exploitable weaknesses in most of the attack surfaces were misconfigurations of the related component.

★ = Caused By Misconfiguration

Name	Date	Attack Surface	Property Violated
Homebrew Jenkins Breach [35]	August, 2018	AS1: Platform ★	Integrity
StackOverflow Teamcity [36]	May, 2019	AS1: Platform ★	Confidentiality (Code, PII)
Webmin RCE [37]	July, 2019	AS1: Platform ★	Integrity
PHP Git Breach [38]	March, 2021	AS1: Platform ★	Integrity
CodeCov Supply Chain Attack [2]	April, 2021	AS2: Container	Integrity
PyTorch [3]	January, 2024	AS4: Runner ★	Integrity
Solar Winds SunBurst [1]	December, 2020	AS5: Pipeline ★	Integrity
Google Flank [39]	February, 2024	AS5: Pipeline ★	Confidentiality (Keys)
Azure Carpenter [40]	August, 2024	AS5: Pipeline ★	Confidentiality (Keys)
ultralitics pypi cryptojacking [25]	December, 2024	AS5: Pipeline ★	Integrity
Kong Ingress Controller Crypto Mining [41]	January, 2025	AS5: Pipeline ★	Integrity
tj-actions changed files supply chain attack [21]	March, 2025	AS6: Provider	Confidentiality (Keys)
GitHub action-download-artifact Poisoning [42]	December, 2022	AS6: Provider	Integrity
Uber GitHub AWS Exfiltration [43]	2014 & 2016	AS6: Provider ★	Confidentiality (PII)
Samsung SmartThings Leak [44]	May, 2019	AS6: Provider ★	Confidentiality (Code)
Mercedes Source Code Leak [45]	May, 2020	AS6: Provider ★	Confidentiality (Code)
Nissan Source Code Leak [46]	January, 2021	AS6: Provider ★	Confidentiality (Code)
New York State IT Leak [47]	June, 2021	AS6: Provider ★	Confidentiality (Code)
DeepSource [48]	July, 2020	AS6: Provider ★	Confidentiality (Code)
Travis CI [49]	September, 2021	AS6: Provider	Confidentiality (Keys)
solana dogwiftool trojan [50]	January, 2025	AS6: Provider	Integrity

able packages; when packaged as artifacts, can contain leaked secrets or credentials.

- **AS3) The Supply Chain:**

All dependencies, trusted or not, consumed or used by the CI/CD pipeline; includes the pipeline’s platform, software from package managers, and external repositories; often a superset of other attack surfaces with added context or complexity.

- **AS4) The Runner:**

The agent that executes arbitrary code specified by pipeline stages; intentions of code, benign or malicious, are not known to the runner; often configured with parameters given by providers.

- **AS5) The Pipeline:**

The declarative configuration of a pipeline, typically written in yaml, determines the permissions, execution environment, stages, etc. about a given pipeline; this attack surface includes pipelines that do not utilize proper security validation or testing.

- **AS6) The Provider:**

Misconfigurations or unknowingly vulnerable capabilities originating from the CI/CD enterprise service itself; includes vulnerabilities within the SCM or VCS.

With our analysis of CI/CD architecture revealing 6 distinct security targets, it is important to understand which of these targets are abused, and how, in real world CI/CD attacks, in order to aid our understanding of necessary security tooling.

V. ATTACKS ON CI/CD PIPELINES

To build a collection of reputable CI/CD attacks, we conducted a manual systemic search using public search engines and academic paper aggregators (e.g., Google Scholar, ACM

DL, IEEE Xplore) for targeted keywords like “CI/CD (attacks)”, “DevOps (attacks)”, and “pipelines”. Our inclusion criteria for selected attacks required that each attack both had a clear description of how it was conducted and which mechanisms were targeted, and was reported on by one or more credible sources (e.g., the GitHub security advisory, OWASP [51], academic papers).

We then aggregated and analyzed these CI/CD related attacks into Table II to understand their origin and impact. Through manual analysis, we classified the targeted attack surface and violated security property for each attack. For the purposes of this work, we focused on attacks targeting the integrity and confidentiality of CI/CD systems. While the availability of CI/CD systems is important, a compromise in the availability of a CI/CD pipeline is often transient and operational, falling outside the scope of our study.

Integrity Violations are characterized by compromised pipelines serving malicious artifacts. Once an attacker has access to a CI/CD system, they can embed untrusted code inside an artifact through a variety of methods, causing a previously benign pipeline to begin producing compromised artifacts. For our purposes, we refer to such attacks as “Artifact Infection”. In most cases, these pipelines are treated as trusted sources by their dependents, causing a propagation of the vulnerable artifact into many reliant systems. The most infamous examples of such outcomes are the SolarWinds Orion attack [1], and the CodeCov supply chain attack [2].

Confidentiality Violations do not directly tamper with pipeline artifacts, but instead exfiltrate sensitive data like source code or private keys outside the breached environment. As such, we refer to attacks violating the confidentiality of a pipeline as “Secret Exfiltration”. The three biggest targets of

secret exfiltration are private keys [21], [39], [40], personally identifiable information (PII) [36], [43], and source code [44]–[47]. Of these three, secret exfiltration of private keys like pipeline tokens and 3rd party API secrets (typically stored in the pipeline for deployments) poses the most significant security threat. Attackers that can access the private keys of a pipeline can use them to perform malicious actions like push contaminated artifacts to package registries on behalf of the pipeline, or use the keys to traverse further into a compromised network. While these three targets are not a comprehensive list of every possible secret, they are representative of common targets we observed in our study of real-world attacks.

Instances of these violations can be high impact, as the centralized nature of pipeline build contexts require them to contain a high number of typically overly permissive credentials and tokens, which can be used by attackers to move laterally inside of compromised environments. Additionally, the prevalence of automated artifact building and distribution means that many dependent pieces of software rely on upstream packages produced by pipelines. An infectious compromise of one upstream package can quickly propagate downstream, spreading rapidly. PaPs, like those supplied by the GitHub Actions Marketplace, make this threat more important to address as pipelines increasingly rely on a common set of reusable components.

After investigating and classifying the 21 attacks in Table II, we note several important observations. Nine of the 21 attacks were achieved through provider bugs and misconfigurations (AS6), five targeted the pipeline (AS5), four focused on the platform (AS1), one on the runner (AS4) and one on the container (AS2). Of these attacks, 16/21 (76%) were caused by misconfigurations. The higher proportion of configuration based attacks illustrates the importance of configuration validation tools capable of validating both the platform and the pipeline configuration files themselves. Given these attack patterns, we will examine state of the art of open source CI/CD security tools to understand how they might address existing vulnerabilities.

VI. EXISTING SECURITY TOOLS

Many tools attempt to address different aspects of securing CI/CD environments, with differing levels of success. Some tools scan for CVEs and validate IaC across the entire platform, like Trivy [64] and Snyk [65], while others like Step Security’s Wait For Secrets [67] tackle smaller challenges like providing multifactor authentication for CI/CD.

In order to address RQ3, we aggregated popular open source CI/CD security tools using systemic keyword searches on public search engines and code hosting sites like GitHub, looking for tools which exported keywords like “GitHub Actions (security)” and “CI/CD (security)”. We then categorized each tool by its advertised purpose, platform support, and popularity.

A. Pipeline Configuration Validation

Pipeline configuration validation (PCV) tools are among the most popular CI/CD security tools. Due to the complicated nature of interconnecting pipeline stages, and limited knowledge of the supply chain at execution time, these tools do not fully guarantee pipeline security (AS5), but do provide useful best practice security advice to avoid common syntax based misconfigurations like command injection. These tools typically come in the form of command line scanners, like codeql extractor iac [32], octoscan [34], github actionlint [31], and zizmor [33]. Most configuration validation tools are capable of identifying syntactic issues and error-prone patterns like improper escaping or environment variable usage. While limited by their corpus of known attacks, these tools can help mitigate vulnerabilities like command injection before they are exploited.

B. Platform and Provider Configuration Validation

Platform and provider configuration validation (PPCV) tools provide recommendations for best practice configuration of both the platform, the provider, and other tertiary software installed in the CI/CD system. Many tools are multipurpose, capable of doing platform configuration validation in addition to pipeline configuration validation and other IaC scanning. Tools like Step Security’s Secure Repo [59] validate provider security configurations, while other tools like trivy [64] and snyk [65] provide insight into Kubernetes and miscellaneous IaC targets. In addition to using these tools to secure AS1 and AS6, practices like network segregation, comprehensive logging, and the usage of intrusion detection or prevention tools are all recommended to secure the CI/CD platform.

Furthermore, providers themselves also have a number of built-in security measures to reduce the attack surface. One such measure is the automatic redacting of secrets from public facing runner logs in order to avoid leaking privileged credentials. In addition, built-in Static Application Security Testing (SAST) is offered by providers like GitHub and GitLab [68], [69] to test and audit code during or separately from CI/CD jobs. Providers also commonly segregate pipeline execution permissions based on the authority of the individual who initiated them, which helps protect pipelines against attacks that abuse vectors like malicious fork PRs.











































C. Software Bill of Materials

Software Bill of Materials (SBOM) generation tools perform comprehensive scanning and digestion to create itemized lists of the exact versioned software installed in an environment or contained within an artifact. These resulting SBOMs can be used to check for the introduction of vulnerable dependencies over time, or cataloged to better understand the TCB of a product. These tools are capable of scanning platform packages (AS1), and tools like syft [56], Trivy [64], and grype [54] are commonly used to perform this on container images (AS2). SBOM tools also provide valuable insight into securing the supply chain (AS3), informing best practices like package pinning.

TABLE III

CI/CD Pipeline Security Tools and their Advertised Functions. These tools should verify the security of the pipeline’s structure and environment, rather than the pipeline artifacts (as plugins).

 GitHub Actions
  GitLab Pipelines
  CircleCI
  Azure Bicep
  ArgoCD
  Terraform
  AWS CloudFormation
  BitBucket Pipelines
  Containers
  Docker Compose
  Helm
  Kubernetes
  Jenkins

Tool	PPCV	PCV	SBOM	SCA	KVD	RAD	Reach	Platform Support
Checkov [52]	✓	✓		✓	✓		7,200 stars on GitHub	    
Codeql Extractor IaC [32]	✓	✓					42 stars on GitHub	    
Cycode cimon [29]						✓	92 stars on GitHub	  
Cycode Raven [53]			✓	✓			635 stars on GitHub	
grype [54]			✓	✓	✓		9,000 stars on GitHub	 OS pkgs, code
kntrl [55]						✓	89 stars on GitHub	
syft [56]			✓				6,400 stars on GitHub	
clair [57]				✓	✓		10,400 stars on GitHub	
cijail [28]						✓	36 stars on GitHub	 
ggshield [58]	✓						1,700 stars on GitHub	Generic
Step Security Harden Runner [30]						✓	638 stars on GitHub	
Step Security Secure Repo [59]		✓					263 stars on GitHub	
Black Duck Detect [60]				✓	✓		164 stars on GitHub	Generic
Veracode SCA [61]				✓	✓		8 stars on GitHub	 (plugin)
Terrascan [62]	✓						4,800 stars on GitHub	 (plugin)  (plugin) 
safedep vet [63]				✓	✓		244 stars on GitHub	 (plugin)  (plugin)  (plugin)
github actionlint [31]		✓					2,900 stars on GitHub	
Trivy [64]	✓		✓	✓	✓		24,100 stars on GitHub	   
Snyk [65]	✓			✓			5,000 stars on GitHub	  
OWASP Dependency Check [66]				✓	✓		6,600 stars on GitHub	  
octoscan [34]		✓			✓		183 stars on GitHub	
zizmor [33]	✓	✓					1,900 stars on GitHub	

D. Software Composition Analysis

Similar to Software Bill of Materials generation, Software Composition Analysis performs deep and comprehensive scanning to understand what components make up a system or artifact, and how they are conjoined. Typically, software composition analysis is a step in the greater process of SBOM generation or KVD, however, there are applications of software composition analysis which are standalone. A good example of this is Cycode’s Raven Analyzer [53], which was created to understand dependency chains between public GitHub Actions pipelines. This tool ingests a great amount of detail about public facing pipelines, including their dependencies, but performs its own unique processing and serialization. SCA tools can be applied to audit and protect AS1, AS2, and AS3.

E. Known Vulnerability Detection

KVD combines the techniques used in SBOM generation and SCA, using the resulting composition information to search known databases for vulnerabilities that exist in presently installed software versions. A KVD tool is often used to perform scans on artifacts like container images, in order to detect if any CVEs or reported vulnerabilities are lurking in system packages [54], [57]. Similar to SBOM and SCA, KVD also defends AS1, AS2, and AS3.

F. Runtime Anomaly Detection

Runtime anomaly detection (RAD) tools are a newer class of CI/CD defense tools, protecting AS4, that assume inherently untrusted execution of CI/CD pipelines. A common approach to increase insight into runners has been to bootstrap monitors inside their execution contexts, reporting on and refusing anomalous network traffic [28]–[30]. In addition to network egress monitoring, some of these runtime defense tools utilize eBPF (Extended Berkeley Packet Filter) to monitor system calls for anomalies [55]. Other novel approaches have been experimented with like two factor authentication during the runner execution via Step Security’s Wait For Secrets [67], but instances like these are mostly experimental and rarely deployed in the wild. This class of tools also includes services like network and egress monitors, which run concurrently on the platform to detect network anomalies.

G. The State of Tools

The categorization of existing CI/CD security tools in Table III has highlighted the importance of many existing cybersecurity practices. As a result of the large breadth of technologies that make up CI/CD and the disjoint nature of tools meant to secure them, collections or bundles of tools have emerged as a leading security solution. Cloud Native Application Protection Platforms (CNAPPs) are collections of tools rolled into one package, typically targeted to run in parallel to the Kubernetes or cloud environments which host the CI/CD stack. These tools

allow for IaC scanning, network analysis, Known Vulnerability Detection (KVD), and more.

Outside of these cloud tools, the open source community has begun developing parsing tools which can analyze pipeline configuration files for misconfigurations, but many do not go beyond syntactic pattern matching for known bad practices. A few dynamic runtime solutions have been developed like the Harden Runner [30], cijail [28], and cimon [29], but with a focus on network anomaly detection, these tools lack detection capabilities for more complicated supply chain attacks which can often pass through the pipeline undetected, like in the case of Artifact Infection.

Given our previously discerned attack classifications, in combination with our aggregation and classification of these CI/CD security tools, we can see a clear relationship between CI/CD attacks and the defense tools which seek to mitigate them. The lower proportion of RAD tooling can explain the presence of attacks targeting AS5, as these breaches are not easily detectable by the more prevalent static configuration analysis and KVD scanning tools. It is imperative that this overlap be explored in order to understand how current security tooling might not be keeping up with CI/CD attacks and to inform the creation of novel security tooling.

VII. GAPS IN THE SECURITY LANDSCAPE

To further examine this challenge, as described in RQ4, we carefully constructed 7 vulnerable or faulty GitHub Actions workflows of differing classes, and tested 5 popular open source security solutions on them to see if they could find or mitigate the embedded errors. Of the security solutions, we tested four static analysis or linting tools, and one runtime security solution. We chose these 5 tools specifically in order to evaluate configuration based vulnerabilities, due to our identification of the prevalence of misconfiguration attacks in Table II. Additionally, we selected these 5 specific tools because they had a low barrier of entry, and do not require payment to evaluate. The source code for our evaluations is publicly available on GitHub¹, along with documentation for each experiment.

As shown in Table IV, each of our 5 selected tools were able to detect a simple command injection which resulted in an anomalous outbound network request, but when the syntax of the command injection was slightly modified (Complex Command Injection), Checkov [52] failed to detect the command injection pattern due to it is regex making assumptions about whitespace. None of the linting tools were able to detect a malicious network request, while the runtime defense tool cimon [29] was able to.

Outside of these 3 positive detections, none of the remaining four pipeline vulnerabilities were detected by any of the five tools we evaluated. Leaking secrets to GitHub runner logs was attempted with three differing levels of complexity, but remained undetected by the security tools in each case. While GitHub itself has built-in secret redaction for plaintext

and base64 encoded pipeline tokens, attacks like tj-actions changed-files [21] where pipeline secrets were exfiltrated by circumventing GitHub log redaction through double base64 encoding, illustrate the importance of transcending trivial pattern matching and syntax checks. Overall, while able to detect simple syntax patterns which enable issues like command injection, the static parsing based tools could not identify any complex or nuanced vulnerabilities in the CI/CD yaml.

Many static linting tools alert to unpinned dependencies in the CI/CD pipeline yaml, but do not have deep insight into the pipeline itself. To illustrate this, we designed the Use Unpinned Dependency evaluator. This evaluation contains a pipeline that builds a C binary with CMake, relying on an external GitHub repository to vendor a library which is used by the binary. In the CMakeLists.txt build file (Listing 1), we reference the external repository without a pinned commit ref, and as such pull in a faulty version of the dependency into the pipeline. This faulty library code is compiled into the artifact, and upon execution in a testing stage, causes the binary to dump the CI/CD runner's environment variables. This is a great example of the lack of insight SOTA tools actually have into the greater execution context of CI/CD pipelines, as only a binary analysis tool would be able to detect such a change in the artifact.

```
cmake_minimum_required(VERSION 3.14)
project(eval)

include(FetchContent)

FetchContent_Declare(
    library

    GIT_REPOSITORY
    https://github.com/riversideresearch/
    CICD_SoK_Evaluators.git

    GIT_TAG main # unpinned ref
)

FetchContent_MakeAvailable(library)

add_executable(eval main.c)
target_link_libraries(eval PRIVATE library)
```

Listing 1: "Use Unpinned Dependency" CMakeLists.txt. SOTA CI/CD tools are unable to reason about an embedded build system like this, even though it directly impacts the CI/CD environment and end-state artifact.

The remaining undetected evaluators, Install Known Bad Dependency and Pin A Bad Dependency, involve a runner installing a log4j vulnerable package, and a pipeline using a pinned version of a known vulnerable version of Harden Runner [30].

¹https://github.com/riversideresearch/CICD_SoK_Evaluators.git

Evaluator	Logging Secret Leak	Anomalous Request	Simple Command Injection	Complex Command Injection	Install Known Bad Dependency	Use Unpinned Dependency	Pin A Bad Dependency
cimon [29]	✗	✓	✓	✓	✗	✗	-
Checkov [52]	✗	✗	✓	✗	✗	✗	✗
github actionlint [31]	✗	✗	✓	✓	✗	✗	✗
octoscan [34]	✗	✗	✓	✓	✗	✗	✗
zizmor [33]	✗	✗	✓	✓	✗	✗	✗

✓ Properly detected fault ✗ Failed to detect fault - Not in scope for tool

TABLE IV

GitHub Vulnerability Detection Evaluation for Existing Tools. SOTA static and dynamic tools are unable to sufficiently reason about the CI/CD pipeline behavior and system at large.

VIII. DISCUSSION & FUTURE WORK

In our analysis of documented CI/CD attacks, we observed that 50% of attacks targeted the CI/CD provider, and 76% involved some form of misconfiguration abuse as its initial attack vector. We then tested 5 open-source CI/CD security tools on 7 evaluation pipelines which contained misconfiguration related vulnerabilities to observe the relevance of our findings first-hand. In our evaluation, vulnerabilities were mitigated by the tools in only 29% of the trials, and 4 out of the 7 evaluations went completely unmitigated by any of the tools.

Current syntactic linting and parsing tools can catch many trivial vulnerabilities, but not more advanced and obfuscated ones. Our study revealed two substantiating examples:

1. While GitHub itself will redact plaintext and base64 encoded pipeline secrets from log files, attacks like tj-actions changed-files [21] have successfully defeated these protections by leaking secrets which were encoded in base64 twice. Existing GitHub Actions configuration linting tools do not alert to log secret leakage in any form, including plaintext.
2. We found a bug in the open source tool Checkov, where the linter’s regular expression for matching command injection patterns erroneously matched against exclusively whitespace characters, ignoring newlines, within the brackets of the embedded template command. Using this knowledge, we were able to successfully perform command injection undetected by this tool through mixing in newline characters.

The commonality between these examples is that syntax-based rules that examine only the form of the configuration commands instead of the run-time (semantic) implications are prone to incompleteness. Therefore, we recommend that future research pursue the creation of semantically aware tooling to conduct complete or property-based testing. These methodologies have been shown in traditional program analysis to be sufficient to syntactic linting in detecting known and unknown forms of vulnerabilities [70].

In addition, the high proportion of attacks targeting CI/CD providers in our findings illustrates a need for greater provider hardening. In particular, we recommend introducing provider-level anomaly detection and supply chain monitoring. CI/CD providers already track and log events as they occur, but too often these event logs are only used after-the-fact to understand how an attack happened, rather than prevent it in real-time. To give an example from the same tj-actions changed-files attack [21], providers should immediately alert to an anomaly when an event like rewriting repository tag references occurs, and lock down a repository to prevent further abuse. While such features could be implemented by providers themselves, they could just as well be implemented through the providers exposed APIs as a standalone service.

Another under-explored front in provider security is supply chain monitoring. Many providers like GitHub also manage a PaP marketplace (like the GitHub Actions marketplace) but don’t actively employ malware scanning techniques on the packages they host. In addition to applying best-practices like malware scanning, these marketplaces would benefit from principled approaches beyond human review for keeping zero days in the supply chain from infecting downstream projects. For instance, providers which own their PaP distribution ecosystem could explore automated version rollbacks and mitigations to stop vulnerabilities in real-time as they are detected and classified in databases like the GitHub Security Advisory.

IX. CONCLUSION

We have presented a novel quantification of CI/CD, including a breakdown of terminology as well as a generalization of its architecture and patterns to aid in understanding the scope of its security needs. Additionally, we have aggregated and categorized 21 in-the-wild CI/CD attacks, identified 22 relevant security tools spanning 6 advertised security capabilities, as well as performed novel gap analysis using 5 of the tools with GitHub Actions pipelines. As a result, we conclude that state-of-the-art CI/CD defense tools are

inadequate for defending against attacks mimicking a fraction of the complexity of those we have recorded in-the-wild, and that the research community should pursue solutions to CI/CD security which emphasize semantic configuration validation as well as provider hardening and anomaly detection.

We have presented a breakdown and generalization of CI/CD terminology, architecture, and patterns to aid in understanding the scope of CI/CD security needs. We aggregated and categorized 21 in-the-wild CI/CD attacks, identified 22 relevant security tools spanning six advertised security capabilities, and performed novel gap analysis using five of the tools with GitHub Actions pipelines. We identified that 76% of our cataloged attacks are based on misconfiguration exploits, often targeting the CI/CD provider service itself, and that state-of-the-art defense tools based on syntactic checks are inadequate for defending against them. We suggest that the research community pursue solutions to CI/CD security that emphasize semantic configuration validation as well as provider hardening and anomaly detection.

X. ACKNOWLEDGEMENTS

The authors thank Dr. Mike Shields and Josh Arey of Vigilant Systems, as well as our colleague Drew Haker, for their assistance in supporting and reviewing this work. This research was supported by US government research funding. The views expressed are those of the authors and do not necessarily reflect the official policy or position of the Department of the Air Force, the Department of Defense, or the U.S. government. Approved for public release; distribution is unlimited 96TW-2025-0024.

REFERENCES

- [1] SolarWinds, "Faq: Security advisory," <https://www.solarwinds.com/sa-overview/securityadvisory/faq>, 2021.
- [2] CodeCov, "Bash uploader security update," <https://about.codecov.io/security-update/?ref=blog.gitguardian.com>, 2021.
- [3] J. Stawinski, "Playing with fire - how we executed a critical supply chain attack on pytorch," <https://johnstawinski.com/2024/01/11/playing-with-fire-how-we-executed-a-critical-supply-chain-attack-on-pytorch/>, 2024.
- [4] H. O. Delicheh and T. Mens, "Mitigating security issues in github actions," in *Proceedings of the 2024 ACM/IEEE 4th International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS) and 2024 IEEE/ACM Second International Workshop on Software Vulnerability*, ser. EnCyCriS/SVM '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 6–11. [Online]. Available: <https://doi.org/10.1145/3643662.3643961>
- [5] H. O. Delicheh, A. Decan, and T. Mens, "Quantifying security issues in reusable javascript actions in github workflows," in *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, 2024, pp. 692–703.
- [6] Z. Pan, W. Shen, X. Wang, Y. Yang, R. Chang, Y. Liu, C. Liu, Y. Liu, and K. Ren, "Ambush from all sides: Understanding security threats in open-source software ci/cd pipelines," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 1, pp. 403–418, 2024.
- [7] J. Ayala and J. Garcia, "An empirical study on workflows and security policies in popular github repositories," in *2023 IEEE/ACM 1st International Workshop on Software Vulnerability (SVM)*. IEEE, May 2023, p. 6–9. [Online]. Available: <http://dx.doi.org/10.1109/SVM59160.2023.00006>
- [8] N. M K, M. B S, N. Khandelwal, N. Pai, and S. L, "Ci/cd pipeline with vulnerability mitigation," in *2023 International Conference on Recent Advances in Science and Engineering Technology (ICRASET)*, 2023, pp. 1–6.
- [9] E. P. Enoiu, D. Truscan, A. Sadovykh, and W. Mallouli, "Veridevops software methodology: Security verification and validation for devops practices," in *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ser. ARES '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: <https://doi.org/10.1145/3600160.3605054>
- [10] L. Nikolov and A. Aleksieva-Petrova, "Framework for integrating threat modeling into a devops pipeline for enhanced software development," in *2024 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 2024, pp. 1–5.
- [11] D. V. Landuyt, L. Sion, W. Philips, and W. Joosen, "From automation to ci/cd: a comparative evaluation of threat modeling tools," in *2024 IEEE Secure Development Conference (SecDev)*, 2024, pp. 35–45.
- [12] A. R. Patel and S. Tyagi, "The state of test automation in devops: A systematic literature review," in *Proceedings of the 2022 Fourteenth International Conference on Contemporary Computing*, ser. IC3-2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 689–695. [Online]. Available: <https://doi.org/10.1145/3549206.3549321>
- [13] T. Klooster, F. Turkmen, G. Broenink, R. T. Hove, and M. Böhme, "Continuous fuzzing: A study of the effectiveness and scalability of fuzzing in ci/cd pipelines," in *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, 2023, pp. 25–32.
- [14] A. Sharma, C. Cadar, and J. Metzman, "Effective fuzzing within ci/cd pipelines (registered report)," in *Proceedings of the 3rd ACM International Fuzzing Workshop*, ser. FUZZING 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 52–60. [Online]. Available: <https://doi.org/10.1145/3678722.3685534>
- [15] Z. Sun, Z. Quan, S. Yu, L. Zhang, and D. Mao, "A knowledge-driven framework for software supply chain security analysis," in *Proceedings of the 2024 8th International Conference on Control Engineering and Artificial Intelligence*, ser. CCEAI '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 267–272. [Online]. Available: <https://doi.org/10.1145/3640824.3640866>
- [16] C. Okafor, T. R. Schorlemmer, S. Torres-Arias, and J. C. Davis, "Sok: Analysis of software supply chain security by establishing secure design properties," in *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, ser. SCORED'22. New York, NY, USA: Association for Computing Machinery, 2022, p. 15–24. [Online]. Available: <https://doi.org/10.1145/3560835.3564556>
- [17] K. Brady, S. Moon, T. Nguyen, and J. Coffman, "Docker container security in cloud computing," in *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, 2020, pp. 0975–0980.
- [18] L. Verderame, L. Caviglione, R. Carbone, and A. Merlo, "Secco: Automated services to secure containers in the devops paradigm," in *Proceedings of the 2023 International Conference on Research in Adaptive and Convergent Systems*, ser. RACS '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: <https://doi.org/10.1145/3599957.3606222>
- [19] A. El Khairi, M. Caselli, C. Knierim, A. Peter, and A. Continella, "Contextualizing system calls in containers for anomaly-based intrusion detection," in *Proceedings of the 2022 on Cloud Computing Security Workshop*, ser. CCSW'22. New York, NY, USA: Association for Computing Machinery, 2022, p. 9–21. [Online]. Available: <https://doi.org/10.1145/3560810.3564266>
- [20] Docker, "Docker setup buildx action," <https://github.com/docker/setup-buildx-action>, 2025.
- [21] StepSecurity, "tj-actions changed-files through 45.0.7 allows remote attackers to discover secrets by reading actions logs," <https://github.com/advisories/GHSA-mrrh-fwg8-r2c3>, 2025.
- [22] A. Khan, "The monsters in your build cache - github actions cache poisoning," <https://adnanthekhan.com/2024/05/06/the-monsters-in-your-build-cache-github-actions-cache-poisoning/>, 2024.
- [23] GitHub, "Github actions cache," <https://github.com/actions/cache>, 2025.
- [24] Y. Gu, L. Ying, H. Chai, Y. Pu, H. Duan, and X. Gao, "More haste, less speed: Cache related security threats in continuous integration services," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 1179–1197.
- [25] GitHub, "Discrepancy between what's in github and what's been published to pypi for v8.3.41," <https://github.com/ultralytics/ultralytics/issues/18027>, 2024.
- [26] —, "Actions runner," <https://github.com/actions/runner>, 2025.

- [27] GitLab, "Run your ci/cd jobs in docker containers," https://docs.gitlab.com/ci/docker/using_docker_images/, 2025.
- [28] Staex, "cijail," <https://github.com/staex-io/cijail>, 2024.
- [29] C. Labs, "Cimon," <https://github.com/CycodeLabs/cimon-action>, 2024.
- [30] S. Security, "Harden runner," <https://github.com/step-security/harden-runner>, 2024.
- [31] rhyds, "actionlint," <https://github.com/rhyds/actionlint>, 2024.
- [32] G. A. Security, "codeql extractor iac," <https://github.com/advanced-security/codeql-extractor-iac>, 2024.
- [33] woodruffw, "zizmor," <https://github.com/woodruffw/zizmor>, 2025.
- [34] synacktiv, "octoscan," <https://github.com/synacktiv/octoscan>, 2025.
- [35] M. McQuaid, "Security incident disclosure," <https://brew.sh/2018/08/05/security-incident-disclosure/>, 2018.
- [36] D. Ward, "A deeper dive into our may 2019 security incident," <https://stackoverflow.blog/2021/01/25/a-deeper-dive-into-our-may-2019-security-incident/>, 2019.
- [37] webmin, "Remote command execution [cve-2019-15231]," <https://webmin.com/security/#remote-command-execution-cve-2019-15231>, 2019.
- [38] N. Popov, "Changes to git commit workflow," <https://news-web.php.net/php.internals/113838>, 2021.
- [39] StepSecurity, "Stepsecurity detects ci/cd supply chain attack in google's open-source project flank in real-time," <https://www.stepsecurity.io/case-studies/flank>, 2024.
- [40] —, "Stepsecurity detects ci/cd supply chain attack in microsoft's open-source project azure karpenter provider in real-time," <https://www.stepsecurity.io/case-studies/azure-karpenter-provider>, 2024.
- [41] Kong, "December 2024 unauthorized kong ingress controller 3.4.0 build," <https://konghq.com/blog/product-releases/december-2024-unauthorized-kong-ingress-controller-3-4-0-build>, 2025.
- [42] woodruffw, "Artifact poisoning vulnerability in action-download-artifact v5 and earlier," <https://github.com/advisories/GHSA-5xr6-xhww-33m4>, 2022.
- [43] M. Gaddy, "Uber breaches," <https://www.breaches.cloud/incidents/uber/>, 2023.
- [44] Z. Whittaker, "Samsung spilled smartthings app source code and secret keys," <https://techcrunch.com/2019/05/08/samsung-source-code-leak/>, 2019.
- [45] C. Cimpanu, "Mercedes-benz onboard logic unit (olu) source code leaks online," <https://www.zdnet.com/article/mercedes-benz-onboard-logic-unit-olu-source-code-leaks-online/>, 2020.
- [46] —, "Nissan source code leaked online after git repo misconfiguration," <https://www.zdnet.com/article/nissan-source-code-leaked-online-after-git-repo-misconfiguration/>, 2021.
- [47] Z. Whittaker, "An internal code repo used by new york state's it office was exposed online," <https://techcrunch.com/2021/06/24/an-internal-code-repo-used-by-new-york-states-it-office-was-exposed-online/>, 2021.
- [48] jai, "Security incident on deepsource's github application," <https://discuss.deepsources.com/t/security-incident-on-deepsources-github-application/131>, 2020.
- [49] Montana, "Security bulletin," <https://travis-ci.community/t/security-bulletin/12081>, 2021.
- [50] B. Computer, "Solana pump.fun tool dogwiftool compromised to drain wallets," <https://www.bleepingcomputer.com/news/security/solana-pumpfun-tool-dogwiftool-compromised-to-drain-wallets/>, 2025.
- [51] GitHub, "Owasp top 10 ci/cd security risks," <https://github.com/cider-security-research/top-10-cid-security-risks>, 2023.
- [52] bridgecrew, "checkov," <https://github.com/bridgecrewio/checkov>, 2024.
- [53] C. Labs, "Raven," <https://github.com/CycodeLabs/raven>, 2024.
- [54] A. Inc, "grype," <https://github.com/anchore/grype>, 2024.
- [55] kondukto io, "kntrl," <https://github.com/kondukto-io/kntrl>, 2025.
- [56] A. Inc, "syft," <https://github.com/anchore/syft>, 2024.
- [57] R. H. Quay, "clair," <https://github.com/quay/clair>, 2024.
- [58] G. Gaurdian, "ggshield," <https://github.com/GitGuardian/ggshield>, 2024.
- [59] S. Security, "Secure repo," <https://github.com/step-security/secure-repo>, 2024.
- [60] B. D. Software, "detect," <https://github.com/blackducksoftware/detect>, 2024.
- [61] V. Software, "Veracode sca," <https://github.com/veracode/veracode-sca>, 2024.
- [62] Tenable, "terrascan," <https://github.com/tenable/terrascan>, 2024.
- [63] SafeDep, "vet," <https://github.com/safedep/vet>, 2024.
- [64] A. Security, "trivy," <https://github.com/aquasecurity/trivy>, 2024.
- [65] Synk, "snyk cli," <https://github.com/snyk/cli>, 2024.
- [66] J. Long, "Owasp dependency check," <https://github.com/jeremylong/DependencyCheck>, 2024.
- [67] S. Security, "Wait for secrets," <https://github.com/step-security/wait-for-secrets>, 2024.
- [68] GitHub, "Github advanced security," <https://github.com/security/advanced-security>, 2025.
- [69] GitLab, "Gitlab static application security testing," https://docs.gitlab.com/user/application_security/sast/, 2025.
- [70] M. Chiari, M. De Pascalis, and M. Pradella, "Static analysis of infrastructure as code: a survey," in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, 2022, pp. 218–225.