A Kubernetes CI/CD Pipeline with Asylo as a Trusted Execution Environment Abstraction Framework

Jamal Mahboob,*† Joel Coffman*‡
* Engineering for Professionals, Whiting School of Engineering, Johns Hopkins University
Email: { jmahboo1, joel.coffman }@jhu.edu
† Google, LLC
‡ Department of Computer and Cyber Sciences, United States Air Force Academy

2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC) | 978-1-6654-1490-6/21/$31.00 ©2021 IEEE | DOI: 10.1109/CCWC51732.2021.9376148

***Abstract*—Modern commercial software development organizations frequently prescribe to a development and deployment pattern for releases known as continuous integration / continuous deployment (CI/CD). Kubernetes, a cluster-based distributed application platform, is often used to implement this pattern. While the abstract concept is fairly well understood, CI/CD implementations vary widely. Resources are scattered across on-premise and cloud-based services, and systems may not be fully automated. Additionally, while a development pipeline may aim to ensure the security of the finished artifact, said artifact may not be protected from outside observers or cloud providers during execution.**

This paper describes a complete CI/CD pipeline running on Kubernetes that addresses four gaps in existing implementations. First, the pipeline supports strong separation-of-duties, partitioning development, security, and operations (i.e., DevSecOps) roles. Second, automation reduces the need for a human interface. Third, resources are scoped to a Kubernetes cluster for portability across environments (e.g., public cloud providers). Fourth, deployment artifacts are secured with Asylo, a development framework for trusted execution environments (TEEs).

***Index Terms*—continuous integration / continuous deployment (CI/CD), Kubernetes, trusted execution environment (TEE), DevSecOps, Asylo, Tekton, Kaniko, Harbor**

I. Introduction

As applications have become increasingly complex, the security of both the finished deployment artifact and the development pipeline that produces that artifact has garnered increased scrutiny. Many new paradigms have been adopted by the software development community, chief among them the introduction of security into the "DevOps" framework, resulting in the "DevSecOps" moniker [1]. DevSecOps, a portmanteau of development, security, and operations, represents the fusion of best practices from these three disciplines with a particular focus on automation.

These systems create automated or semi-automated "pipelines" whereby both source code and build artifacts (e.g., compiled executables) are scanned by various processes in order to ensure their overall security against known attack vectors. Modern industry software development and deployment pipelines can be broadly broken down into a two-stage process: continuous integration (CI) followed by continuous deployment (CD). CI traditionally uses automated testing to ensure that changes do not break anything—i.e., the code compiles and passes its test suite [2]. More recently, this process is augmented with additional checks, including peer review, lint tools that enforce readability and adherence to a coding standard, static analysis for known critical vulnerabilities (CVEs), and dynamic analysis of build artifacts for security or performance issues. The result of the CI stage is a deployment artifact, such as a compiled executable or an *image*, a read-only template for creating *containers* managed by an operating system kernel. Following the successful completion of the CI stage, a second CD stage deploys the artifact into production [3]. These CD processes generally do not address the security of artifacts at run-time, particularly their confidentiality when stored on or executed by third-party infrastructure (e.g., a public cloud provider). With the introduction of multiple competing infrastructure, platform, and software "as-a-service" providers, the number of additional outside parties in which an organization must trust has increased. To reduce the footprint of trusted external entities as much as possible, our CI/CD pipeline scopes as many resources within Kubernetes[1] as possible. Our design has the added benefit of enabling the pipeline to be portable across multiple service providers and infrastructure capabilities.

Pipeline and artifact security is not limited to artifacts however. None of the aforementioned systems, nor the pipelines that execute them, protect against an attacker with administrative privileges (e.g., an insider threat) [4]. Such threats include personnel employed outside an organization, such as those employed by a cloud service provider. Consequently, processor manufacturers introduced TEEs into their architectures. TEEs are a processor-based technology whereby code "in-process" is secured by the hardware itself through instructions that encrypt and secure memory and page tables [5], [6]. The most popular and prevalent of these is Intel's Software Guard Extensions (SGX) [7], which introduces new security-related instructions into the computing base. This hardware support allows a running binary to be opaque to an attacker with

[1] https://kubernetes.io/

978-0-7381-4394-1/21/$31.00 ©2021 IEEE

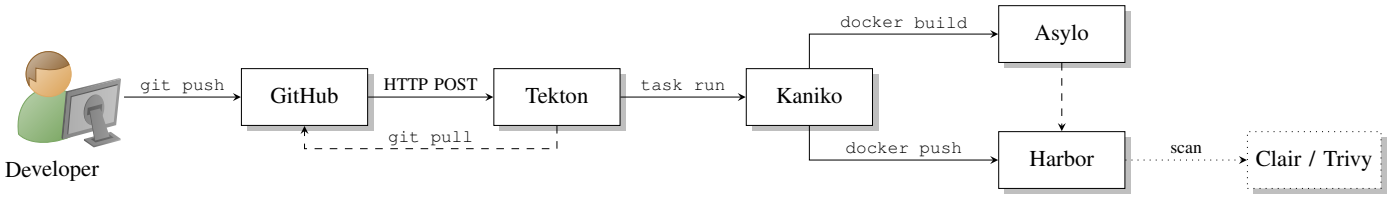

Fig. 1. The CI stage of our two-stage pipeline. When a developer pushes to the remote repository, the pipeline automatically builds and pushes a finished artifact to the container registry (Harbor). Scanning by Clair / Trivy has been shown in prior work; we do not implement it in our prototype.

administrative privileges on the system itself. Unfortunately, adoption is challenging due to the requirement for application developers to rewrite existing code. Asylo offers an easy-to-use system as a "wrapper" around code to enable developers to make use of TEEs without the need to re-architect their code bases. This paper demonstrates that it is possible to extend a CI/CD pipeline with an additional step that ensures a finished binary can execute in a secure enclave.

The contributions of our work are as follows:

- Our CI/CD pipeline is architected using Kubernetes, an increasingly popular Platform as a Service (PaaS) offering, and can be migrated trivially to different cloud providers or to on-premise infrastructure.
- We extend the CI/CD pipeline with an additional step that automatically deploys artifacts using Asylo, a TEE framework, to ensure the confidentiality of deployment artifacts.
- We provide a comprehensive evaluation that focuses on the security, performance, and limitations of our CI/CD pipeline.

To the best of our knowledge, this work is the first to describe a comparable CI/CD pipeline in detail, particularly one that is portable across cloud service providers. Furthermore, we have released our implementation as an open source project to allow others to extend our work.[2]

The remainder of this paper is organized as follows. Section II provides an overview of our CI/CD pipeline. Section III describes the pipeline's implementation using specific technologies and services. We evaluate our work in Section IV. Section V highlights related work, and we conclude in Section VI.

II. Design

This paper describes a repeatable, portable, and secure CI/CD pipeline scoped locally to a Kubernetes cluster. Kubernetes is a open-source capability for application development and deployment in a large-scale, clustered, distributed system [8]. While an open-source project, Kubernetes is governed by the Cloud Native Computing Foundation (CNCF) which uses a variety of steering committees and special interest groups to determine the life-cycle and future feature set of the system. Kubernetes is best described as a "platform," fitting into the PaaS framework of cloud computing technologies (Infrastructure as a Service (IaaS), PaaS, Software as a Service (SaaS), etc). While many major cloud providers today (e.g., Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform) provide managed Kubernetes-as-a-Service offerings, it is also possible for individuals and organizations to run a Kubernetes cluster in their own on-premise systems, using a commercial cloud provider's IaaS offering, or on purpose-built offerings such as those designated for use by governments.

Many enterprise organizations are concerned with the overall security of their software development pipeline and finished artifacts, as well as the overall repeatability and portability of their solutions [9] so as to avoid vendor lock-in [10]. Tekton, an open source framework for creating CI/CD pipelines, abstracts implementation details and standardizes CI/CD across environments (e.g., multiple cloud providers) and tools. By utilizing Kubernetes and Tekton, our CI/CD pipeline is inherently portable across cloud providers and even to on-premise or air-gapped networks. There are many cloud service providers that make Kubernetes available "as-a-Service" (Google, Amazon, and Microsoft to name a few) on which this pipeline could be deployed. Additionally, on-premise bare-metal Kubernetes clusters can be provisioned either with open-source software such as `kops` or `kubeadm` or through commercial offerings (e.g., Pivotal Kubernetes Service or Rancher Kubernetes Engine). With only a top-level requirement of Kubernetes as a platform layer, this pipeline is replicable on all of these systems.

This section describes the two stages of the pipeline (CI and CD) and how the final artifact is built and deployed using Asylo. The use of a TEE for deployed artifacts uniquely separates this pipeline from prior work (e.g., [11]–[13]).

A. Continuous Integration

As depicted in Figure 1, a software developer initiates the CI pipeline by committing and sharing changes (i.e., pushing to a remote repository). Following this action, event listeners automate each successive action in the pipeline, from building the artifact to storing it for future use. Tekton automates each of these actions. More specifically, a Tekton event listener checks out the latest commit (which was just pushed by the developer) and builds an image using a `Dockerfile` in the repository. Building the image is performed by Kaniko, a tool to build containers inside a Kubernetes cluster, using Asylo to ensure the confidentiality of the image. Upon success, the finished artifact (i.e., the image) is pushed into Harbor, a container registry scoped locally to the Kubernetes cluster. If all these actions complete successfully, the second pipeline stage (i.e.,

[2]https://github.com/google/cluster-scoped-cicd

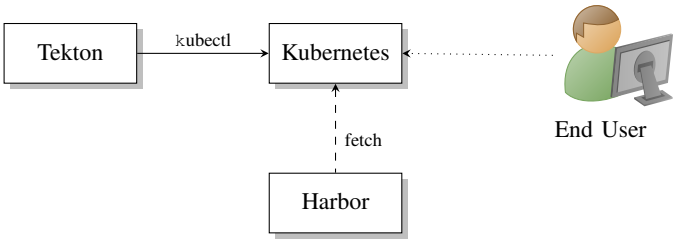

Fig. 2. The CD stage of our two-stage pipeline. Tekton triggers a second action and artifacts previously published to Harbor will be pulled and (re)deployed to our production namespace.

CD stage) begins execution immediately; otherwise, the CI pipeline fails, denotes the cause of the failure (i.e., the action that failed), and stops execution.

B. Continuous Deployment

Definitions differ when referring to a CD pipeline with many organizations distinguishing "delivery" from "deployment" where the latter refers to the automation of the actual deployment of finished artifacts to production [14], [15]. Using "continuous delivery" as the definition, we could stop at placing the artifact into a container registry. This definition represents only the CI stage of our pipeline. Instead, our CD stage differentiates our pipeline from prior work (e.g., [12]) in that finished artifacts are deployed to running production systems.

After the successful completion of the CI stage, Tekton initiates the CD stage with a reference to the latest artifact pushed to Harbor. This workflow either initiates an initial deployment of the artifact to the Kubernetes cluster or redeploys of the artifact over the top of the currently running or previous deployment. Figure 2 depicts this workflow in detail.

C. Trusted Execution Environment

To utilize a TEE (i.e., "enclave"), source code must be compiled to use the requisite Instruction Set Architecture (ISA) extensions and be executed on a compatible processor [16]. A binary not compiled (in the CI stage) with these instructions cannot utilize the underlying execution environment when deployed. Additionally, a binary built to invoke these instructions will fail deployment (the CD stage) when invoking instructions on a processor without the requisite ISA extensions.

To ensure the portability of software executing on the pipeline, and not just the portability of the pipeline itself, a TEE abstraction framework, Asylo, was used to develop C++ code for execution. Our pipeline makes use of `Dockerfiles` to invoke a pre-built Asylo build environment container. If the deployment environment does not support Intel SGX, the build environment container can create a new container with an embedded simulator to execute the binary.

III. Implementation

We use a variety of open-source software and commercial cloud services to implement our pipeline. As many of these components as possible are deployed on a Kubernetes cluster managed using Google Kubernetes Engine (GKE). GKE is a managed "Kubernetes-as-a-Service" capability provided by Google Cloud Platform. Once provisioned, a user has a CNCF-compliant Kubernetes cluster with master nodes managed by the provider (i.e., Google Cloud Platform) and worker nodes / cluster versions determined by the user. We use GitHub for version control, which is not cluster-scoped, but alternatives such as GitLab can be used to integrate cluster-scoped version control into our pipeline.

The remainder of this section describes the implementation of our CI/CD pipeline in detail. We walk through each service in the order in which it is invoked as depicted in Figures 1 and 2—i.e., GitHub, Tekton, Kaniko, Asylo, and Harbor.

A. GitHub

GitHub[3] is a service popular with software developers that exposes a way to centrally host Git repositories online. Individual developers and enterprise organizations expose code repositories and version control their assets publicly or privately, allowing for various developer-centric workflows. While GitHub is a broad ecosystem, only a few critical components are used in our CI pipeline.

We host a private Git repository using GitHub. An access token allows systems to authenticate and be authorized to perform repository actions. Using this token, a web-hook sends an `HTTP POST` event to the URL endpoint of a Tekton event listener whenever a `git push` command is sent to the `master` branch of the repository (see Figure 1). The event listener triggers the CI pipeline upon receiving the event.

B. Tekton

Tekton[4] comprises four components for CI/CD workflows: pipelines, triggers, a command-line interface (CLI), and dashboard for visualization. A *pipeline* defines the core functionality of Tekton. A pipeline is a list of tasks, each of which comprises one or more steps. Each task executes in a Kubernetes *pod*, a group of containers that are co-located and co-scheduled by Kubernetes, and each step executes in a separate container within that pod. Tekton *triggers* define event listeners to execute pipelines. We do not use the CLI or dashboard for our CI/CD pipeline.

1) Tekton Triggers: Tekton Triggers is an open-source event trigger library specifically architected to work with Kubernetes. Tekton Triggers creates new Kubernetes Custom Resource Definitions (CRDs) to expose `TriggerTemplate`, `TriggerBinding`, and `EventListener` objects that can be used by Kubernetes deployment systems.

For our pipeline, we create a single `EventListener` object that accepts `HTTP POST` events from an external source (i.e., GitHub). The event listener extracts the repository reference(s) from the payload, passing the inputs along to Tekton Pipelines. Once this event has been validated, Tekton Triggers begins execution of a `PipelineRun` action to start our CI/CD pipeline.

[3]https://github.com/
[4]https://tekton.dev/

2) Tekton Pipelines: Tekton Pipelines is an additional component in the Tekton ecosystem. Much like Tekton Triggers, Tekton Pipelines exposes new CRDs to the Kubernetes API that can be made available by other components of the system. Tekton Pipelines exposes new `Task`, `TaskRun`,[5] `Pipeline`, and `PipelineRun` CRDs.

The pipeline is constructed out of a single `Pipeline` that consists of two `Task` actions. When the Tekton Triggers `EventListener` receives an event (i.e., an HTTP POST request), it instantiates a `PipelineRun` that begins an execution of the `Pipeline` given the input and output targets extracted from the event's payload. The `Pipeline` performs a checkout of the required code and builds it using the provided `Dockerfile` using Kaniko as an execution container and Asylo as a binary wrapper. The resulting artifact from this `Task` is a Docker image, which is pushed to Harbor, a container registry scoped locally to the Kubernetes cluster. If any of the `Task`'s steps fail, the pipeline fails, denotes the step at which it failed, and ceases execution.

After the successful completion of the CI pipeline, Tekton Pipelines triggers an additional `TaskRun`. This `Task` receives a reference the latest artifact (i.e., Docker image) that was pushed to Harbor and triggers either an initial deployment of that artifact to the Kubernetes cluster or a re-deployment of that artifact over the top of the currently running or previous deployment. This CD stage differentiates our CI/CD pipeline from prior work (e.g., [12]) in that finished artifacts are deployed to running production systems.

C. Kaniko

Our CI stage uses container artifacts from the open-source Kaniko project.[6] Kaniko differs from the previous Tekton objects: it is not a system of containers that lives as a CRD within the Kubernetes cluster, but a simple container artifact that deploys to the cluster as an executing object.

D. Asylo

Asylo[7] is an open-source project that provides a framework to develop binaries that will execute in a processor's secure enclave or TEE. A completed Asylo artifact is a binary that targets a specific processor architecture. This binary is not inherently set-up to execute correctly in the context of a container or a Kubernetes cluster. Consequently, Asylo provides a simulation environment (using `--config-sgx-sim`) for testing. This environment removes the dependency on Intel SGX-capable hardware for development and experimentation.

For our pipeline, the CI stage results in the creation of a finished container artifact containing an Asylo-wrapped binary. More specifically, the Asylo-wrapped binary comprises a new container in which the binary executes the Intel SGX simulation environment. This container is then deployed as an artifact to the Kubernetes cluster.

E. Harbor

Kubernetes requires that containers to be deployed to the cluster come from a known and secure container registry. The most popular framework for this is Docker Hub, a publicly-available registry provided by Docker. Harbor[8] allows organizations to deploy and to secure their own private registry. Third-party CVE scanners such as Clair[9] and Trivy[10] can be integrated into Harbor to scan images and to restrict their use if violations are discovered (a capability not available in Docker Hub).

The final step of our CI pipeline stores an image in Harbor. This action then either deploys the image to the Kubernetes cluster or redeploys the image in the currently running or previous deployment.

IV. Evaluation

This section details our evaluation. Our primary focus is the security of the CI/CD pipeline through least privilege and the security of artifacts by specifically targeting a TEE for the artifacts' execution. We also discuss automation and portability before highlighting the performance of our pipeline. Finally, this section concludes with several limitations, which suggest opportunities for future work.

A. Separation of Duties

One of the core tenets of modern security practices is the concept of least privilege [17]. This principle ensures that each component—human or machine—in a system is only permitted to perform the exact actions necessary to complete the task(s) that they have been given. For instance, a machine that is supposed to process objects and write them to a storage system does not inherently require permission to read objects from that same storage system. Consequently, if the credentials from the processing system are stolen, then an attacker would still only be able to write new objects to the storage system and would not be able to use their credentials to read, download, or otherwise steal the objects already present.

The core component of our pipeline is the development aspect of DevSecOps.

Dev (Developer) These individuals are responsible for writing code. While they may be able to develop, test, and deploy locally in test environments, they should not have permission to access a staging or production environment. Instead, they push revised or newly developed code into a version control system.

Sec (Security) These individuals are responsible for attesting to the security of the code and deployment artifacts in the system. They are responsible for neither developing new code nor ensuring production systems maintain up-time targets.

[5]It is not necessary for `TaskRun` actions to be explicitly instantiated as part of a `Pipeline`. The tasks defined in a `Pipeline` dynamically create `TaskRun` actions when a `PipelineRun` is created. However, `TaskRun` actions are still available for implicit execution for development and testing purposes.

[6]https://github.com/GoogleContainerTools/kaniko

[7]https://asylo.dev/

[8]https://goharbor.io/

[9]https://github.com/quay/clair

[10]https://github.com/aquasecurity/trivy

Ops (Operations) These individuals are responsible for maintaining finished, secured artifacts in production. In addition, they are responsible for the stability of the production system itself and for the availability and integrity of the running production artifacts.

Our CI/CD pipeline enforces strong separation of duties [18], each with principles of least privilege. Developers have only the permission to `push` and `pull` code into the version control system. They are not responsible (nor able) to assume permissions to attest to the security of their artifacts nor can they abuse production systems. Security is handled almost entirely through code automation. Security personnel define the scanning utilities in Harbor and set acceptable vulnerability targets (e.g., Critical, High, Medium, etc.) that allow finished artifacts to be pulled from Harbor and deployed into production systems. Once defined, these systems execute automatically (see Section IV-C). Finally, pushing finished artifacts to operations is handled automatically. Only operations personnel have the ability to touch the production Kubernetes cluster.

B. Securing Deployed Artifacts

Traditional security components of software development pipelines are concerned with securing the finished artifacts against attacks from outside observers. For example, Harbor automatically scans for known vulnerabilities and exploits (CVEs) that may have been introduced either by the code or the base container artifacts themselves.

An often overlooked security threat stems from the infrastructure provider. Consequently, processor manufacturers introduced secure enclaves or TEEs in order to secure code "in-process" from even the infrastructure provider themselves. Being able to secure data and data processing systems in motion adds an additional security item to organizations currently securing data in transit and at rest. By wrapping code artifacts with Asylo in the CI stage of the pipeline, deployment of those artifacts can ensure execution in a secure enclave in the CD stage.

C. Automation

The automation of processes through the pipeline reduces the number of manual processes involved in the deployment of secure software and contributes greatly to the repeatability of the pipeline. Each code change undergoes the exact same integration and deployment stages.

Without the automation inherent in the pipeline, the following five steps must be performed manually by the appropriate human personnel:

1) Check code in to a version control system
2) Build container artifacts based on the source code
3) Tag and push the container artifact to a container registry
4) Scan the container artifact for CVEs
5) Deploy the finished artifact to production

With separation of duties and automation, the pipeline has been reduced to a single step, a code check-in. Once code is checked in to version control, no human intervention is necessary to ensure a finished, secured artifact is deployed to production.

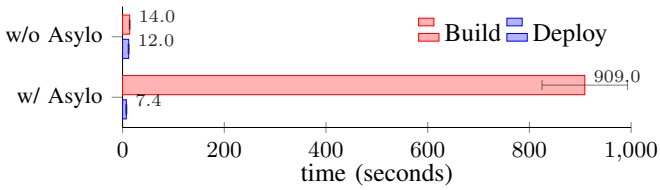

Fig. 3. Mean execution times for the CI/CD pipeline without and with Asylo. Error bars indicate the standard deviation.

D. Cluster-Scoped Resources

Of particular concern for organizations is the desire to have portable and repeatable solutions. Organizations must be able to repeat the software development process for either a new application or a new business sector, and be able to move solutions to other cloud providers, on-premise networks, or even air-gapped systems.

1) Repeatability: Our pipeline is architected with Kubernetes, an increasingly popular PaaS capability. Kubernetes evangelizes both software and infrastructure repeatability through the concept of "infrastructure as code."

2) Portability: Enterprise organizations have the need to support a variety of customers and mission owners in a variety of different domains. Our pipeline scopes as many resources as possible locally within the cluster, allowing organizations to levy a single technical requirement for their solution, the presence of CNCF-compliant Kubernetes infrastructure on which to deploy their software. An organization may develop their software stack with this pipeline on Kubernetes infrastructure provided by the PaaS capabilities of one cloud service provider and then migrate the entire application and development stack to another cloud service provider in order to meet customer demand, pricing structure changes, availability, etc. Some customers may also have regulatory requirements that prescribe data processing be performed locally with on-premise infrastructure. Our pipeline can be prototyped and developed in Kubernetes infrastructure provided by a cloud service provider and then migrated to an on-premise network to meet this requirement. Certain privacy sensitive customers such as governments or defense organizations may have the need to deploy software on air-gapped systems not connected to global Internet infrastructure. Our pipeline can be migrated to completely air-gapped networks to accommodate these requirements.

E. Performance

We examined the performance of our pipeline, focusing on the overhead introduced when packaging the artifact with Asylo. As CI/CD is already practiced by many software development organizations, Asylo represents the major novelty of our work apart from our CI/CD pipeline's portability. Our experiments used the Google Compute Engine (GCE) machine type of n1-standard-4 with 4 vCPUs, 15 GB RAM, and traditional magnetic disks.

The Asylo container object is large (more than 2 GB) and must compile against a simulated Intel SGX processor

environment. This requirement creates an additional stage that introduces significant delay to the pipeline. We configured additional event listeners to trigger two pipeline runs in parallel, one with and one without Asylo, and record the difference between the first and last event's timestamp for each pipeline run. Figure 3 depicts the time spent for the build (i.e., CI) and deploy (i.e., CD) stages.

Without Asylo as a final execution stage, our CI/CD pipeline completes in an average of 26.0 seconds. This time is roughly equally spent between building and deploying an artifact. Introducing Asylo increases the build time by almost two orders of magnitude, resulting in an average successful pipeline execution time of 916.4 seconds. We attribute this increase to the size and computational complexity introduced by Asylo's Intel SGX simulation environment. This increase does not extend to deployment where the timing is comparable to when Asylo is not used. Nevertheless, the Kubernetes worker nodes running the build could be significantly increased in power to reduce the time when using Asylo.

F. Limitations

Our existing implementation has several open source dependencies that currently have shortcomings. We discuss these issues in the remainder of this section.

The use of TEEs greatly improves the security of finished artifacts executing in production systems. However, TEEs are only introduced in the final "Ops" section of the pipeline. A cloud service provider administrator or other external malicious entity still has the ability to inspect components as they move through the "Dev" and "Sec" components of the system. An attacker may be able to inspect resources as they sit in Harbor, for instance.

Currently Asylo only supports Intel SGX as an underlying technology, though its road map lists other TEEs such as AMD Secure Encrypted Virtualization (SEV) and ARM TrustZone. Furthermore, the Asylo project, along with many TEE environments, require developers to use C/C++ as a base language. The introduction of support for additional compiled languages also appears on the road map for the Asylo project.

We developed, tested, and evaluated our pipeline using Asylo's `sgx-sim` capability, which allows developers to prototype solutions without needing access to potentially expensive new hardware. The lack of test and evaluation of the pipeline on actual Intel SGX hardware is an item for future work.

V. Related Work

The use of CI/CD is increasingly common for software development. Existing work combines these tools and services in unique ways, leading to custom CI/CD pipelines tied to specific vendors (e.g., [13]). Aravena [11] describes GitOps and a variety of tools for a Kubernetes CI/CD pipeline. We use many of these tools in our work, providing both a concrete implementation and a detailed evaluation of our pipeline. Brady et al. [12] describe a CI pipeline for Docker images. Our work not only includes deployment of artifacts to a Kubernetes cluster (i.e., CD) but also applies to multiple types of artifacts instead of simply Docker images. Buchanan et al. [13] describe a CI/CD pipeline running in the Azure Kubernetes Service. Though their work is scoped entirely within Azure, it is not designed for portability; in contrast, our CI/CD pipeline uses open source projects to allow migration across cloud service providers. None of these prior efforts consider the security of the artifact at run-time as we do with Asylo.

Despite the introduction of ARM TrustZone in 2004, Intel SGX in 2015, and AMD SEV in 2019, these TEEs are not yet widely used [6]. Our use of Asylo proves that it is feasible to extend a CI/CD pipeline with an additional step so that artifacts are protected against untrusted administrators, including those of a cloud service provider, without requiring extensive changes to applications. Though many alternatives techniques exist (see Patil and Modi [19] for a survey), our integration into a CI/CD pipeline minimizes an existing barrier to adoption.

VI. Conclusion

Our CI/CD pipeline, running nearly entirely within a Kubernetes cluster, introduces the ability for binaries to be deployed using containers to a Kubernetes cluster that supports secure enclaves provided by TEEs. Unlike prior work, we explicitly address the CD stage and the security of artifacts from untrusted system administrators, including those of the cloud provider. Scoping our CI/CD locally within the Kubernetes cluster increases the overall security of the system by reducing the number of outside, external entities that must be trusted. It has the added benefit of contributing to both the repeatability and portability of the overall system, a capability desired by many enterprise organizations for a variety of security and business purposes.

The use of GitHub as a managed, external version control system is an existing gap in the desire to scope resources entirely within a Kubernetes cluster. We intend to explore alternatives, such as GitLab, to overcome this limitation. In addition, our implementation and evaluation was done entirely on Google Cloud Platform. Further work is required to demonstrate the repeatability and portability of our CI/CD pipeline using another cloud service provider, but we are confident that our extensive use of open source projects will minimize the need for any changes when porting our pipeline to another environment. Finally, integrating third-party CVE scanners such as Clair or Trivy remain a straightforward task for future work.

With the ultimate goal of security of both the pipeline and deployed artifacts, additional future efforts in alternative attack vectors and their mitigations, such as a moving target defense, can be explored. Nomad [20] co-opts the ability for cloud service providers to perform a live migration of a virtual machine (VM) to mitigate side channel attacks, and MIGRATE [21] extends this idea to containers, migrating them among physical hosts. The portable nature of our work may make it possible to migrate workloads not only between hosts within a cloud provider, but between cloud providers as well.

ACKNOWLEDGMENTS

We thank the members of Google Cloud's Global Public Sector (GPS) Customer Engineering team for their input on the initial direction of the project.

REFERENCES

[1] N. Tomas, J. Li, and H. Huang, "An empirical study on culture, automation, measurement, and sharing of devsecops," in *2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*. New York, NY: IEEE, 2019, pp. 1–8.

[2] M. Meyer, "Continuous Integration and Its Tools," *IEEE Software*, vol. 31, no. 3, pp. 14–16, 2014.

[3] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Upper Saddle River, NJ: Addison-Wesley Professional, 2010.

[4] W. R. Claycomb and A. Nicoll, "Insider Threats to Cloud Computing: Directions for New Research Challenges," in *2012 IEEE 36th Annual Computer Software and Applications Conference*. New York, NY: IEEE, 2012, pp. 387–394.

[5] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted Execution Environment: What It is, and What It is Not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1. New York, NY: IEEE, 2015, pp. 57–64.

[6] P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Trusted Execution Environments: Properties, Applications, and Challenges," *IEEE Security & Privacy*, vol. 18, no. 2, pp. 56–60, 2020.

[7] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '13. New York, NY: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2487726.2488368

[8] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade," *Queue*, vol. 14, no. 1, p. 70–93, January 2016. [Online]. Available: https://doi.org/10.1145/2898442.2898444

[9] P. Perera, R. Silva, and I. Perera, "Improve Software Quality through Practicing DevOps," in *2017 Seventeenth International Conference on Advances in ICT for Emerging Regions (ICTer)*. New York, NY: IEEE, 2017, pp. 1–6.

[10] J. Opara-Martins, R. Sahandi, and F. Tian, "Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective," *Journal of Cloud Computing*, vol. 5, no. 4, 2016.

[11] R. Aravena, "Let Your Software Supply Chain Ride with Kubernetes CI/CD," in *Proceedings of the 33rd Large Installation System Administration Conference*, ser. LISA '19. Berkeley, CA: USENIX Association, October 2019.

[12] 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*, ser. CCWC '20. New York, NY: IEEE, January 2020, pp. 975–980.

[13] S. Buchanan, J. Rangama, and N. Bellavance, *CI/CD with Azure Kubernetes Service*. Berkeley, CA: Apress, 2020, pp. 191–219.

[14] Pittet, "Continuous integration vs. continuous delivery vs. continuous deployment," https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment, 2020.

[15] Anastasov, "What's the difference between continuous integration, continuous delivery, and continuous deployment," https://semaphoreci.com/blog/2017/07/27/what-is-the-difference-between-continuous-integration-continuous-deployment-and-continuous-delivery.html, July 2017.

[16] V. J. Zimmer, P. J. Barry, R. Poornachandran, A. V. D. Ven, P. A. Dice, G. Selvaraje, J. Carreno, and L. G. Rosenbaum, "Providing a trusted execution environment using a processor," 2014, uS9594927B2.

[17] F. B. Schneider, "Least privilege and more [computer security]," *IEEE Security Privacy*, vol. 1, no. 5, pp. 55–59, 2003.

[18] R. Sandhu, "Separation of duties in computerized information systems," Proc. of the IFIP WG11.3 Workshop on Database Security, September 1990.

[19] R. Patil and C. Modi, "An Exhaustive Survey on Security Concerns and Solutions at Different Components of Virtualization," *ACM Computing Surveys*, vol. 52, no. 1, pp. 12:1–12:38, February 2019. [Online]. Available: https://doi.org/10.1145/3287306

[20] S.-J. Moon, V. Sekar, and M. K. Reiter, "Nomad: Mitigating Arbitrary Cloud Side Channels via Provider-Assisted Migration," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1595–1606. [Online]. Available: https://doi.org/10.1145/2810103.2813706

[21] M. Azab and M. Eltoweissy, "MIGRATE: Towards a Lightweight Moving-Target Defense Against Cloud Side-Channels," in *2016 IEEE Security and Privacy Workshops (SPW)*, 2016, pp. 96–103.