A cryptographically verified approach to secure Docker container updates in CI/CD pipelines*

Vitaliy Tymoshchuk[1,†], Dmytro Tymoshchuk[1,*,†], Mykola Mytnyk[1,†], Yurii Klots[2,†], and Nataliia Petliak[2,†]

[1] *Ternopil Ivan Puluj National Technical University, 56 Ruska str., 46001 Ternopil, Ukraine*
[2] *Khmelnytskyi National University, 11 Instytutska str., 29016 Khmelnytskyi, Ukraine*

Abstract

The article investigates the problem of secure updates of containerized applications under automated CI/CD processes. It is shown that traditional deployment of container images does not provide guarantees of artifact authenticity and integrity and increases the risk of software supply chain attacks in the event of compromise of the container registry, network channel, or pipeline configuration. This work presents a secure CI/CD update workflow that combines keyless Cosign signing (OIDC/Fulcio), immutable SHA-256 digest–based image addressing, mandatory pre-deployment signature verification, and freshness controls against rollback attacks, while permitting controlled rollback only to previously verified versions. In addition, signature annotations are employed to bind artifacts to specific commits and workflows, thereby strengthening auditability. The solution is integrated into the CI/CD pipeline without modifying its core logic. Experimental modeling of attack scenarios confirms the prevention of image substitution and a reduction in detection time with negligible overhead associated with signing and verification. The obtained results demonstrate the feasibility of applying the proposed approach in DevSecOps practices.

Keywords

container security, CI/CD pipeline, container image signing, software supply chain, Docker, Cosign, Sigstore, GitHub Actions, container registry.

1. Introduction

Containerization has become a standard approach for application deployment in modern software development due to its flexibility, scalability, and efficiency. At the same time, the widespread adoption of container technologies has made them an attractive target for cybercriminals. The security of containers and their update processes has therefore become critically important, especially in the context of Continuous Integration/Continuous Delivery (CI/CD) pipelines, where updates on servers are performed automatically and continuously. A compromise of container image integrity at any stage of the lifecycle, from code development to deployment, may result in the introduction of vulnerabilities or the injection of malicious code into the production environment. Attackers may attempt to compromise the software supply chain by embedding malicious components or modifying container images during their transfer to the execution environment [1]. According to industry studies, the number of software supply chain attacks has been rapidly increasing and doubled in 2024 compared to the previous period [2].

It is important to note that traditional security mechanisms, including intrusion prevention systems [3-5], firewalls, and other network-based security solutions [6-8], are generally ineffective at detecting such attacks, as compromised artifacts are distributed as legitimate updates and do not exhibit anomalous network behavior during delivery. These limitations of perimeter-based security are consistent with findings from studies on the design of secure authentication, authorization, and accounting services [9], as well as analyses of risks associated with improper secret management in

* *CSDP'2026: Cyber Security and Data Protection, May 7, 2026, Lviv, Ukraine*
* Corresponding author.
† These authors contributed equally.
tymoshchuk@tntu.edu.ua (V. Tymoshchuk); dmytro.tymoshchuk@gmail.com (D. Tymoshchuk); mytnyk@networkacad.net (M. Mytnyk); klots@khmnu.edu.ua (Y. Klots); npetlyak@khmnu.edu.ua (N. Petliak)
0009-0007-2858-9434 (V. Tymoshchuk); 0000-0003-0246-2236 (D. Tymoshchuk); 0000-0003-3743-6310 (M. Mytnyk); 0000-0002-3914-0989 (Y. Klots); 0000-0001-5971-4428 (N. Petliak)

source code [10], which emphasize the need to shift security controls directly into software development and delivery processes. This underscores the relevance of the problem and justifies the necessity of implementing robust cryptographic mechanisms for container protection directly at the update stage.

One of the typical approaches to automated container updates on servers is the configuration of a CI/CD process that builds a new Docker image upon each code update, pushes it to a container registry, and then pulls this image on the target server, often using the latest tag, for execution. However, such an approach does not guarantee the authenticity and integrity of the retrieved image. If an attacker gains access to the container registry or compromises the image delivery path (e. g., via stolen credentials or a compromised trust chain), image substitution may occur, resulting in the deployment of a malicious image instead of a legitimate update. The absence of mechanisms for verifying the authenticity and integrity of container images creates conditions for the stealthy introduction of malicious code into the production environment.

Furthermore, the use of so-called “floating” tags (e. g., latest) without binding to a specific cryptographic image hash complicates version tracking and rollback operations in the event of security incidents. In particular, it becomes difficult to guarantee that the latest tag at a given point in time still refers to the same verified and secure image rather than a potentially compromised one. In traditional approaches to software security, source code analysis and dependency inspection have dominated, whereas with the widespread adoption of container technologies, security priorities are shifting toward ensuring the integrity and authenticity of software artifacts throughout the delivery process. A container image passes through multiple stages of the CI/CD pipeline [1]. The lack of guarantees regarding container image authenticity creates prerequisites for exploiting CI/CD processes as a channel for software supply chain attacks, significantly increasing the risk of compromising production systems. In this context, the development and implementation of models capable of ensuring secure container updates become particularly important and is regarded as a necessary component of robust DevSecOps practices in the modern software industry.

Within the scope of this study, the problem of ensuring secure updates of Docker containers [11] in a server environment under automated CI/CD processes is addressed. The aim of this research is to increase trust in automated container update mechanisms by integrating cryptographic methods for signing container images and verifying their authenticity and integrity directly prior to deployment.

2. Security challenges and mitigations in CI/CD environments

Software supply chain security issues have attracted significant attention from the scientific and professional communities following high-profile incidents, most notably the SolarWinds attack in late 2020 [12]. This attack clearly demonstrated that the compromise of software build or update processes within CI/CD pipelines can lead to large-scale injection of malicious code into end-user information systems.

In the work [13], the authors identified four stages of a supply chain attack and proposed three fundamental security properties — transparency, authenticity, and isolation — required to protect the software supply chain. They analyzed contemporary security approaches, both academic and industrial, and mapped them to these properties, highlighting the strengths and weaknesses of existing solutions as well as gaps in protection against known attacks.

Consequently, effective software supply chain security should ensure process transparency, cryptographic verification of the authenticity of each artifact, and isolation of execution environments, so that none of the CI/CD stages becomes a vulnerable link susceptible to compromise by attackers. Current efforts are therefore focused on strengthening these security aspects by introducing additional layers of control and protection into software development and deployment processes.

Alongside threats within the software supply chain, numerous studies point to the presence of vulnerabilities directly within CI/CD pipelines. Automated pipeline processes typically operate with elevated privileges, including access to source code repositories, secrets, and deployment mechanisms, which, under misconfigured or insufficiently restrictive settings, makes them an attractive target for attacks. In [14], systemic security issues in the GitHub Actions service [15] are demonstrated. By analyzing approximately 447,000 CI/CD workflows across 213,000 repositories, the authors found that 99.8% of GitHub Actions configurations use excessive access privileges, granting read-write permissions instead of the minimally required read-only access to source code. In addition, 23.7% of workflows are triggered by the pull_request event while executing code from user-controlled branches, creating opportunities for malicious code execution through specially crafted pull requests. The study also revealed that 99.7% of projects rely on third-party Action plugins, with 97% of these actions originating from unverified authors, and that 18% of projects include Actions with known vulnerabilities or lacking up-to-date security patches. Taken together, these factors form a broad range of potential attack vectors through which an adversary may compromise a third-party plugin or inject malicious code via a pull request and thereby gain access to the CI/CD environment. The authors emphasize that these issues create real prerequisites for software supply chain attacks in the context of automated CI/CD processes. The results highlight the critical need to enforce strict access control policies, conduct thorough vetting of third-party components, and ensure secure configuration of CI/CD pipelines.

In the context of secure Docker container updates in CI/CD environments, studies focused on container image security are of particular scientific relevance, addressing issues ranging from image hosting and distribution in container registries to the timeliness and reliability of updates. Container image repositories, most notably Docker Hub [16], have become an integral component of modern CI/CD processes; however, they also represent a significant source of security risks.

In [17], the authors conducted a large-scale analysis of the public Docker Hub container registry by collecting and examining data on 2,227,244 container images and their characteristics. The results revealed several critical security issues. First, a substantial number of images were found to use insecure runtime configurations, including excessive privileges or host directory mounts, which may lead to host data leakage or enable denial-of-service (DoS) attacks [18] during container execution. Second, 42 malicious images were identified that contained embedded malware capable of performing remote code execution or exploiting the victim's computational resources for cryptocurrency mining. Third, the analysis showed that the remediation of known vulnerabilities in container images is often delayed or entirely neglected, with many images retaining outdated and vulnerable packages for extended periods. These findings highlight the high risk that both deliberately compromised images and outdated images with unpatched vulnerabilities may enter CI/CD processes through public container registries such as Docker Hub.

The issue of outdated vulnerabilities is examined in greater detail in [19]. The authors assessed the security of 380 container images over the period from July 2022 to January 2023 by analyzing their update dynamics and the accumulation of vulnerabilities. The results showed that, despite regular updates by developers and maintenance teams, the overall number of identified vulnerabilities in container images does not decrease over time and, in some cases, even increases. In particular, it was found that many popular images are based on operating systems characterized by the accumulation of a large number of known vulnerabilities. For example, images based on Debian were shown to be significantly more vulnerable than those built on Alpine or other minimal distributions. Debian-based variants exhibited a substantially higher number of CVEs, some of which dated back to the late 1990s. At the same time, the authors emphasize that not all identified vulnerabilities pose the same level of practical risk. Less than 1% of the recorded vulnerabilities at any given time represented a high real-world risk for containers, meaning that they had available exploits and potentially significant security impact. This finding underscores the need for a risk-oriented approach to vulnerability remediation, in which priority is given to vulnerabilities that are realistically exploitable rather than to the formal pursuit of complete vulnerability elimination.

Taken together, the results of studies [17] and [19] indicate the need to integrate automated container image vulnerability scanning mechanisms into CI/CD processes, as well as to enforce control over base operating system selection and the timely updating of components, which is critical for minimizing risks during automated container updates. To address the described threats, researchers have proposed enhancements to container repository infrastructures. In [20], the authors presented an architecture based on cryptographic data structures designed to establish trust in container images stored in untrusted environments. The proposed Trustworthy Container Repository (TCR) infrastructure employs authenticated data structures, including Merkle trees, along with hardware-based trust modules to guarantee the integrity, availability, and confidentiality of container images even when stored in public or partially compromised repositories.

One of the key methods for enhancing the security of container update processes is the use of cryptographic signing of container images and signature verification directly within the CI/CD pipeline. The signing mechanism makes it possible to guarantee the authenticity and integrity of an image. Prior to deployment, the system can verify that a container image originates from a trusted source and has not been subjected to unauthorized modifications. In addition, a properly designed signing system can prevent rollback attacks, in which an adversary attempts to enforce the deployment of an outdated and potentially vulnerable container image instead of the current one. This is achieved through version control and freshness verification of signed artifacts. As early as 2015, the Docker community introduced the Docker Content Trust mechanism, implemented in the Notary project [21], which is based on The Update Framework (TUF) standard [22]. Notary provides container image signing and version management and is designed to achieve several important security properties, including resilience to key compromise, guarantees of image freshness to protect against the reuse of outdated code, and support for flexible trust models with delegated signing authority. At the same time, as indicated by industry surveys, the adoption level of such mechanisms in industrial CI/CD environments remained low for a long period.

The solution proposed by the authors in [23] aimed to make cryptographic signing of software artifacts, including container images, a widespread and practically feasible practice. The developed Sigstore mechanism was designed with a focus on lowering adoption barriers for developers. Instead of requiring the management of a dedicated public key infrastructure (PKI) or long-lived private keys, the platform enables signature authentication using existing accounts that support the OIDC protocol, such as Google or GitHub services. After successful one-time authentication, Sigstore generates a short-lived (ephemeral) cryptographic key that is used exclusively to sign a specific artifact, thereby eliminating the need for persistent storage and protection of private keys. Information about each signature, as well as its binding to the identified developer identity, is recorded in a public transparency log, which is an append-only registry that allows any party to verify which artifact was signed, by whom, and at what point in time. The introduction of this level of transparency prevents the stealthy concealment of compromised signatures or artifact substitution. Even in the event of private key compromise, any attempt to use the key to sign another artifact would be recorded in the transparency log and can be detected through independent verification.

It is noteworthy that the Sigstore platform has gained broad industry support within a relatively short period of time. In particular, by 2022, more than 2.2 million cryptographic signatures had been generated using Sigstore for critical projects, including Kubernetes [24] and Distroless distributions [25]. These figures demonstrate the practical effectiveness of the proposed approach: the combination of an open transparency infrastructure with minimal key management requirements and a high level of developer usability significantly increases the willingness to integrate cryptographic signing mechanisms into CI/CD processes.

Thus, an analysis of contemporary scientific and applied studies indicates that ensuring the security of container updates in CI/CD environments must be implemented at several complementary levels. First, it is necessary to protect CI/CD processes themselves, in particular by minimizing privileges, eliminating configuration errors, and controlling the use of third-party

components. Second, significant attention should be devoted to container image security, including the use of trusted sources, automated scanning for known vulnerabilities, and timely updating of software dependencies. Third, the implementation of cryptographic trust mechanisms is critically important, specifically container image signing and signature verification prior to deployment. Approaches based on The Update Framework standard, including those implemented in the Notary project, provide cryptographic protection against image tampering and rollback attacks. At the same time, modern services such as Sigstore make these mechanisms practically applicable and scalable by integrating them directly into automated CI/CD processes.

In this way, it becomes possible to form a holistic view of a multi-layer security system that encompasses organizational policies and DevSecOps practices, CI/CD configuration and component control, as well as cryptographic verification mechanisms for software artifacts. The comprehensive implementation of these approaches establishes a foundation for reliable and secure Docker container updates in modern CI/CD environments.

3. Secure CI/CD update workflow

Based on the conducted analysis, it is established that the key challenge lies in guaranteeing the integrity and authenticity of a container image during automated updates, regardless of potential compromises of infrastructure components, including the CI server, container registry, or data transmission network. In this context, there is a need to ensure cryptographically verifiable provenance of the container image and its binding to the build process within the CI/CD pipeline. In other words, the deployment server must be able to reliably verify that a container image was built by an authorized CI environment from a specific version of the source code and was not subjected to unauthorized modifications during delivery. This property is achieved through the use of cryptographic signing mechanisms, whereby the container image is signed with a private key at the build stage and, prior to execution, undergoes signature verification using the corresponding public key (Figure 1).

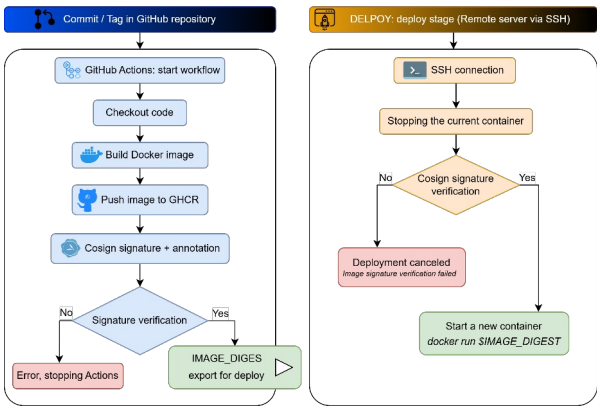

Figure 1: Secure CI/CD workflow for Docker container updates with cryptographic image signing and mandatory image verification.

To eliminate the possibility of unauthorized modification of a container image after the build process has completed, it is necessary to ensure its unambiguous identification based on a cryptographic hash. For this purpose, container images should be transferred and referenced using their cryptographic digest (SHA-256) [26] rather than symbolic tags. The use of digest-based image retrieval guarantees that even if the same tag is retained, the container engine will not accept different content, since any modification of the image results in a change of its hash. Accordingly, CI/CD processes employ container image addressing in the format repository@sha256:<digest> instead of so-called “floating” tags. This approach ensures bit-for-bit equivalence between the retrieved container image and the artifact that was built and signed at the build stage, thereby eliminating the risk of its covert substitution during delivery or deployment.

Verification of the cryptographic signature of a container image must be performed automatically at the deployment stage. If the signature verification fails, either because the image is not signed by an authorized key or because the signature is missing, such an image must not be allowed to be deployed into the production environment. Enforcing this requirement is critically important for protecting against container registry compromise, as even in the event that an attacker attempts to introduce a forged or unsigned image, the system will automatically reject it at deployment time, thereby preventing the execution of untrusted code.

A secure update system must provide detection and prevention of rollback attacks aimed at reintroducing outdated and vulnerable versions of container images under the guise of legitimate updates. Such an attack may occur when an adversary attempts to deploy a container image that was correctly signed in the past but contains vulnerabilities that have already been mitigated in newer versions. To counter such scenarios, an update acceptance policy was implemented that allows deployment only of container images that are unambiguously identified as newer than the currently deployed version. This is achieved by using version information embedded in the cryptographic signature metadata. In addition, signature attributes and signing timestamps are used during verification to assess the freshness of the artifact. Within the scope of this problem formulation, protection against rollback attacks is achieved by combining container image version control with the use of a transparency log for signatures, which provides trustworthy information about which image was signed, by whom, and at what time. At the same time, while preventing rollback attacks is essential, legitimate rollback to a previous version in the event of errors in a new release must remain possible. Accordingly, the secure update system supports controlled and rapid rollback to the last known stable container image. For this purpose, all previous container image versions are retained together with their corresponding cryptographic signatures, and the rollback process itself is subject to mandatory verification. In particular, the system does not allow deployment of a previous container image version if its integrity or authenticity cannot be confirmed. Thus, the rollback mechanism is strictly limited to container images that were previously correctly signed and successfully verified, effectively preventing the introduction of foreign or compromised images during recovery scenarios.

The proposed solution is integrated into the existing CI/CD pipeline as additional signing and verification steps, without modifying the core build and deployment logic. An important requirement is the use of open standards and technologies, in particular Cosign [27] and OCI-compliant container registries, as well as compatibility with widely used automation tools such as GitHub Actions and Docker. In line with DevOps practices focused on maximum automation, the processes of cryptographic signing and verification of container images are fully automated within the CI/CD pipeline. This integration is transparent to users and implemented directly within the pipeline, thereby minimizing operational overhead and reducing the risk of security mechanisms being bypassed due to excessive complexity.

To illustrate the operation of the presented workflow, an experimental scenario of automated updating of a web application running in a Docker container on a remote server was constructed. Within this scenario, a project repository was created on the GitHub platform, a CI/CD pipeline based on GitHub Actions was configured, and a server environment with Docker support was prepared for deploying containerized application updates. During the implementation of the scenario, the full set of security mechanisms was applied, including automated container image signing, cryptographic signature verification prior to deployment, digest-based image addressing, and enforcement of update policies [14]. This practical setup demonstrates the effectiveness of the proposed approach to secure Docker container updates in a CI/CD environment.

The CI/CD pipeline configuration file .github/workflows/ci.yml implements a two-stage (two jobs) automated deployment process consisting of the build_and_sign and deploy stages. The build_and_sign stage is responsible for generating and preparing the container artifact and includes the following sequential steps:

- Building a Docker image of the web application from the repository source code;
- Authenticating to GitHub Container Registry (GHCR) using a CI access token;

- Publishing (pushing) the image to GHCR with a tag derived from the short commit SHA (e.g., ghcr.io/<owner>/<image>:<short_commit_sha>), which unambiguously binds the artifact to a specific source revision;
- Retrieving the image content digest (SHA-256) (e.g., via docker inspect), resulting in an immutable reference of the form ghcr.io/<owner>/<image>@sha256:<digest>.

The obtained digest is subsequently used as the sole identifier of the container artifact, ensuring its immutability and preventing content substitution during subsequent stages of the CI/CD process.

After obtaining the cryptographic digest of the container image, the image is signed using the Cosign tool. The signature is generated directly for the image digest, which ensures an unambiguous binding of the signature to the specific binary content of the container.

An example of the corresponding command is shown below:

```
cosign sign --yes $IMAGE_DIGEST \
  --oidc-issuer https://token.actions.githubusercontent.com \
  --oidc-client-id sigstore \
  --timeout 5m
```

In this scenario, Cosign is used in keyless signing mode, which is based on authentication via the OpenID Connect (OIDC) protocol [28] within the GitHub Actions environment. Instead of managing long-lived private keys, Cosign obtains a short-lived trust certificate from the Fulcio service, which is bound to the authenticated OIDC subject, namely the CI process. This approach significantly reduces the risk of key compromise and simplifies the integration of signing mechanisms into the CI/CD pipeline. The --timeout 5m parameter limits the maximum waiting time for obtaining the OIDC token and completing the signing operation, thereby preventing the CI process from hanging in the event of failures in external services.

To enhance the traceability of container image provenance, a set of metadata annotations is added to the cryptographic signature, capturing the build context within the CI/CD environment. These annotations include the workflow identifier, run number, source code commit hash, and build timestamp, which makes it possible to unambiguously associate the signed artifact with a specific execution of the CI process.

An example of adding such annotations during the signing process is shown below:

```
  --annotations github_workflow="${GITHUB_WORKFLOW}" \
  --annotations github_run_id="${GITHUB_RUN_ID}" \
  --annotations commit="${GITHUB_SHA}" \
  --annotations build_time="$(date -Iseconds)"
```

The specified metadata are embedded into the signature and subsequently recorded in the Sigstore transparency log, providing trustworthy and traceable information about the provenance of the container image. This makes it possible not only to verify the integrity and authenticity of the artifact, but also to determine which CI/CD workflow, based on which commit, and at what point in time was responsible for its creation.

After the container image signing operation is completed within the CI/CD pipeline, immediate signature verification is performed as part of the CI stage. This approach implements a dual-control mechanism that makes it possible to ensure, already at the build stage, that the signature has been correctly applied and that it corresponds to the expected execution context. Verification is performed using the Cosign tool and checks not only the cryptographic validity of the signature, but also the identity of the signing subject, including the correspondence of the Fulcio certificate to a specific repository, CI/CD workflow, and triggering event. An example of the verification command is shown below:

```
cosign verify $IMAGE_DIGEST \
  --certificate-identity-regexp "^https://github.com/${GITHUB_REPOSITORY}/\.github/workflows/ci\.yml@refs/
heads/.+$" \
  --certificate-oidc-issuer "https://token.actions.githubusercontent.com"
```

The --certificate-identity parameter specifies the expected identity of the signing subject, which includes the repository name and the path to the workflow file that initiated the signing process. The --certificate-oidc-issuer option restricts trust to certificates issued exclusively via the GitHub OIDC provider, thereby preventing the use of signatures generated in external or forged environments. If any of the verification conditions fail, the CI/CD pipeline is immediately halted, preventing progression to the deployment stage. This enables early detection of configuration errors or artifact substitution attempts and provides an additional layer of protection for the software supply chain.

After the successful completion of the build_and_sign stage, the container artifact (the image together with its corresponding cryptographic signature) becomes available in the container registry. Subsequently, the deploy stage is initiated, which is responsible for the secure deployment of the update to the server-side execution environment. The deployment stage is executed in a remote environment, which may be implemented via an SSH connection to the server, a dedicated SSH executor provided by GitHub Actions, or an agent of a container orchestration system (e.g., Kubernetes). It is worth noting that the security of the SSH transport layer itself is a subject of ongoing research, particularly in the context of post-quantum cryptographic preparedness [34]. In our experimental setup, deployment was performed using an SSH connection to the server. At this stage, the deployment mechanism receives the IMAGE_DIGEST variable obtained from the previous CI/CD stage, which uniquely identifies the container image to be deployed. Prior to launching the new container, the currently running application instance is stopped to avoid resource conflicts, particularly network port collisions. When using continuous deployment strategies, this step can be adapted to support parallel container execution. A key element of the deploy stage is the verification of the cryptographic signature of the container image immediately before execution. During this verification step, the system retrieves the signature and certificate from the container registry and validates them. Restricting the acceptable signer via the --certificate-identity parameter ensures that the signature is bound to a specific repository and CI/CD workflow. As a result, even if a third-party or malicious project publishes a container image with the same name, its deployment will be rejected because the signer identity does not satisfy the specified constraints. This mechanism enforces the principle of trusting only artifacts produced by an authorized CI/CD process and provides protection against container image substitution at the deployment stage.

If the cryptographic signature verification is successful, the Cosign tool returns a result in JSON format containing information about the signer, the signing timestamp, and the associated annotations. In the event of verification failure, the command terminates with a non-zero exit code, which leads to the immediate termination of the deployment procedure with the error message “Image signature verification failed”. The deployment process configuration explicitly specifies that, under these conditions, the container update must not proceed. After successful signature verification, the new container image is launched. The container is executed directly from the image identified by its cryptographic digest, ensuring bit-for-bit equivalence with the signed artifact. Since the image has passed authenticity and integrity verification, its deployment does not introduce additional security risks.

Upon completion of the deployment, the active container image digest is recorded in the deployment log. This information is used to implement a controlled rollback mechanism, as it enables unambiguous identification of the previous stable version to which the system can revert if necessary. The rollback procedure involves executing the deployment workflow in a special mode, in which a previous image digest is selected from the stored history (Figure 2).

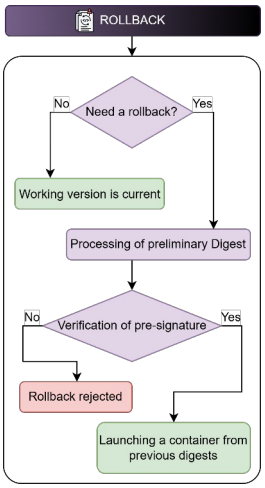

Figure 2: Execution logic of a controlled rollback in a secure CI/CD process.

For the selected image, cryptographic signature verification is performed again, after which the corresponding container is launched. Thus, even in disaster recovery scenarios, rollback is carried out in a controlled manner and in compliance with all trust policies, preventing the circumvention of security mechanisms through rollback operations.

The next section presents the results of an experimental comparison between the baseline and secured CI/CD pipelines, confirming the effectiveness of the proposed approach to secure Docker container updates.

4. Threat modeling and experimental evaluation

To assess the reliability and practical effectiveness of the proposed approach, threat modeling [29] of the CI/CD-based container update system was performed, along with a comparative analysis of attack and failure scenarios for two automated update approaches.

The first approach (baseline) corresponds to common deployment practice, in which the CI/CD pipeline builds a container image, publishes it to a registry, and deploys it on the server using a symbolic tag (e.g., latest) without cryptographic signing or authenticity verification. The second approach (secured) implements container image signing using Cosign, mandatory signature verification prior to deployment, the use of immutable image references based on cryptographic digests, and additional update and rollback control policies [16]. Within the conducted analysis, key security assumptions were validated not at an abstract level, but through the reproduction of typical operational scenarios and adversarial actions characteristic of real-world CI/CD environments. This approach made it possible to demonstrate how the applied protection mechanisms counter specific threats, as well as to delineate the boundaries of their effectiveness under conditions involving the compromise of individual infrastructure components.

In the secured update workflow, a developer introduces changes to the repository source code, after which the CI/CD system automatically builds a container image, generates a unique cryptographic SHA-256 digest for it (denoted as sha256:AAA), and performs cryptographic signing. At the next stage, the deployment pipeline receives the digest value AAA itself rather than a symbolic tag and verifies the signature using Cosign. Provided that the authenticity and integrity verification of the container image is successful, the corresponding container is launched in the production environment. As a result, users receive the updated version of the service, while the deployment logs record the successful transition specifically to the container image identified by digest AAA. This approach demonstrates that the described workflow does not require significant changes to established development and operational workflows. The only additional step is cryptographic verification of the artifact's authenticity prior to execution, ensuring that each accepted update is accompanied by trustworthy and reproducible confirmation of its provenance.

The study also examined typical operational and security violation scenarios related to the update process of containerized applications in CI/CD environments (Appendix A).

In Scenario 1 (container registry compromise), a situation is considered in which an attacker, having obtained unauthorized access to a container registry, attempts to substitute a container image under the tag used for automated deployment in the CI/CD process. In the baseline approach, where deployment is performed using symbolic tags without additional verification, such an attack results in the execution of a forged container and subsequent compromise of the production environment, since the runtime environment trusts the tag as a pointer to the artifact. In the secured architecture proposed in this work, the possibility of such substitution is already constrained at the level of the image addressing mechanism. Specifically, deployment is bound not to a tag, but to a specific cryptographic digest of the container image (IMAGE_DIGEST = sha256:AAA), which is immutable and uniquely identifies the binary content of the artifact.

If an attacker publishes a different container image under the same tag before deployment (Scenario 2), but with a different digest (sha256:BBB), this does not affect the deployment process. At the deployment stage, the system attempts to retrieve the image identified specifically by digest AAA. If this image is unavailable, the docker pull operation fails with an error; if it is available, the original unmodified image is retrieved, since the cryptographic digest is immutable and uniquely determines the binary content of the container image, unlike symbolic tags. Afterward, cryptographic signature verification is performed using Cosign. For the genuine artifact AAA, the signature is valid and the update proceeds normally. Thus, container image substitution through tag manipulation in the registry does not provide an attacker with any practical advantage under the secured approach, as the combination of immutable digest-based addressing and mandatory signature verification effectively neutralizes this attack vector. Under more extreme assumptions, where an attacker attempts to substitute the cryptographic digest of the container image itself, the feasibility of such an attack would require finding a collision for the SHA-256 hash function, which is considered practically infeasible given the current state of computational capabilities.

Another scenario involves a deep compromise of the container registry (Scenario 3), in which, in response to a request for an image identified by digest AAA, the registry returns the byte content of a different artifact (denoted as BBB). In this case, the cryptographic verification mechanism based on Cosign detects the mismatch, since the digital signature is strictly bound to a specific digest and cannot be valid for different content. As a result, the deployment process is aborted and execution of the previously deployed stable container version is preserved. The failure of signature verification serves as a clear indicator of a security incident, enabling timely notification of the operations team and initiation of incident response procedures. From a logical standpoint, this approach mitigates typical incidents in the class of “execution of unauthorized container images due to insufficiently secured runtime environments.” In particular, in scenarios where attackers gain the ability to run containers through a misconfigured Docker API, mandatory cryptographic verification prevents the execution of untrusted images, as they lack a valid signature from an authorized source.

A separate analysis was conducted for a scenario in which an individual with access to the server environment attempts to run a container image outside the approved CI/CD pipeline (Scenario 4) by executing commands such as docker pull …:latest and docker run without prior cryptographic verification. Within the conducted modeling, it was observed that, at the level of the standard Docker mechanism and in the absence of additional control policies, such a container can indeed be executed if signature verification is not explicitly enforced. For this reason, in the proposed approach, this risk is mitigated through an organizational and technical operational policy according to which containerized application updates are performed exclusively via the automated CI/CD pipeline, and cryptographic signature verification is a mandatory prerequisite for deployment. Manual execution of containers in the production environment is not considered an acceptable operational scenario. As an additional layer of security hardening, policy enforcement mechanisms at the runtime level are also considered, including enterprise control solutions or admission mechanisms in Kubernetes that are capable of preventing the execution of container

images lacking the expected cryptographic attributes. Within the scope of this project, such mechanisms were not implemented as mandatory components; however, they logically fit as potential extensions of the proposed architecture. Thus, even in the event of an attempt to bypass the CI/CD pipeline, the subsequent introduction of runtime-level policies can provide an additional security barrier and prevent the execution of untrusted container images.

A rollback scenario (Scenario 5) was analyzed as a separate case, namely a situation in which, following a failed release, there is a legitimate need to revert to a previous stable version of the application. In the reproduced scenario, container image version 1 is identified by digest AAA, whereas version 2 is identified by digest CCC. After a defect is detected in the new release, a rollback to image AAA is initiated. Since the system maintains information about trusted container images and cryptographic signature verification using Cosign is performed again prior to each execution, such a rollback is both correct and secure. Image AAA represents the same signed artifact that previously passed authenticity and integrity checks, and its provenance can be reproducibly verified. In parallel, a malicious rollback attempt was modeled, in which an attacker seeks to enforce a rollback not to the legitimate image AAA, but to a specially crafted container image DDD (for example, an outdated revision or an image containing embedded malicious code). If such an image lacks a valid signature from an authorized source, the Cosign verification mechanism rejects it at the deployment stage.

A more complex scenario (Scenario 6) arises in the event of a compromise of the signing capability, for example, when trust in the signing subject or key material is violated. In such a situation, merely verifying the presence of a valid signature is no longer sufficient, and there is a need to enforce freshness and version control of artifacts. For this purpose, the secured scenario introduces a forward-only policy, according to which version information or build timestamps are recorded in the signature metadata, and the deployment logic, implemented via a dedicated policy engine, prohibits the execution of artifacts whose signatures are chronologically older than that of the currently deployed release. This approach ensures that rollback, as a form of "backward update," is possible only as a deliberate and explicitly authorized decision by the operations team, rather than as an automatically accepted operation. The practical effect of the proposed scheme is that an attacker cannot force the system to accept an outdated, vulnerable, or modified artifact solely on the basis of the formal presence of a signature. The use of a signature log, in particular the Sigstore Rekor transparency log, additionally provides chronological event recording and simplifies auditing of when, by whom, and under what conditions a container image was signed.

An important element of the conducted threat modeling was a clear delineation of the guarantees provided by cryptographic container image signing mechanisms and the threats that lie beyond their scope. From a cryptographic perspective, a classical replay attack in the narrow sense, namely reusing a signature for a different container image, is not practically feasible in the proposed architecture, since the signature is strictly bound to the cryptographic digest (SHA-256) of a specific artifact. Any modification of the image content inevitably results in a change of the digest, rendering the signature invalid. Practically relevant replay-related scenarios are therefore not associated with signature reuse, but rather with attempts to redeploy older yet validly signed container images. Such attacks belong to the class of rollback attacks and are effectively mitigated not by the mere presence of a signature, but by additional control mechanisms, including versioning and freshness policies that prohibit acceptance of artifacts that are chronologically or version-wise older than the currently deployed release.

At the same time, a fundamental limitation of cryptographic signing lies in the injection of malicious code at the build stage. If an attacker gains access to the CI environment, for example through a compromised dependency, runner, secrets, or supply chain component, the artifact may be modified prior to signing. In this case, the container image would be malicious yet formally validly signed, and the cryptographic scheme itself is unable to distinguish it from a legitimate build output. For this reason, within the scope of the analysis, such risks are treated as requiring mitigation by controls operating before the signing stage. These include automated vulnerability scanning of container images (e.g., Trivy, Clair) [30], complemented by security regression

visualization techniques integrated directly into CI/CD workflows [32], hermetic build practices [33], isolation of CI environments, mandatory code review, as well as monitoring of artifacts and attestations in transparency logs, which enables more timely detection of atypical or anomalous signatures and pipeline behavior. A critical risk of signing key compromise was also considered, as theft of a private key would allow an attacker to sign arbitrary images as "trusted." Consequently, a key priority of the presented workflow is the use of the Sigstore keyless strategy, in which long-lived private keys do not exist as a class. Short-lived keys are generated only for the duration of the CI pipeline execution and are bound to the OIDC identity of a specific workflow. When user-managed keys are employed, they must be stored in secure vaults (e. g., Vault) with clearly defined rotation and revocation procedures. Transparency logs further serve as an auditing mechanism, simplifying the detection of anomalous signatures and supporting digital forensic investigations of information security incidents. The modeling also considered availability-related risks during the verification stage, including unavailability of verification services or network failures. Although such incidents do not directly compromise artifact integrity or authenticity, they can affect update availability. In this context, a fail-closed policy is essential. If verification cannot be performed, deployment does not proceed and responsible personnel are notified. Where necessary, a limited offline mode using cached trusted artifacts may be permitted, but without fully bypassing the verification procedure.

To validate the results of the threat modeling, experimental simulations of attacks and failures were conducted for two container update approaches: one without protective mechanisms and one with integrated cryptographic verification controls. Subsequently, a set of key incidents was simulated, including container image substitution, attempts at unauthorized rollback, the presence of vulnerabilities, as well as system behavior during normal update operations and the performance impact (overhead) introduced by the security mechanisms. The results of the comparative simulations are summarized in Table 1.

Analysis of the data presented in Table 1 demonstrates significant advantages of the secured approach compared to the baseline approach. In the baseline CI/CD pipeline, attacks remained undetected at the deployment stage. In one case, a malicious container image was successfully launched in the production environment, and the incident was identified only post factum based on indirect indicators of anomalous activity. In another case, an unauthorized rollback to a vulnerable version was not recognized as an incident, since the system lacked mechanisms for verifying artifact authenticity and freshness. In contrast, under the secured scenario, none of the simulated adversarial actions resulted in compromise. Cryptographic signing and mandatory verification mechanisms for container images enabled immediate detection of inconsistencies, causing the CI/CD pipeline to automatically terminate the deployment process before container execution.

This confirms that integrating authenticity verification at the CI/CD level significantly reduces incident response time: the attack does not progress to the exploitation phase, and subsequent actions are limited to analyzing the attempted violation rather than remediating consequences in the production environment. The obtained results also indicate improved reliability and transparency of the update process. In the secured approach, each deployment was unambiguously linked to a verified source of container image provenance, reducing operational risks and fostering greater confidence in automated updates. Despite the introduction of additional security stages, the impact on overall pipeline duration proved to be minimal. The total time required for signing and verifying a container image of approximately 500 MB was about 25 seconds, which represents an acceptable trade-off for the achieved level of protection.

Overall, the experimental results indicate that introducing container image signing using Cosign, combined with appropriate verification policies, significantly enhances the security of automated Docker container updates in CI/CD environments without a noticeable negative impact on deployment efficiency. A key advantage of the proposed approach lies in the combination of cryptographically grounded trust in container images with a high degree of practical integration into existing CI/CD workflows.

Table 1
Comparative evaluation of baseline and secured CI/CD container update approaches

Metric/Case	Baseline approach (CI/CD without signing)	Secured approach (CI/CD with signing)
Successful updates (no attacks)	3/3 (100%) — updates completed; trust is based solely on infrastructure, without cryptographic confirmation	3/3 (100%) — updates completed; each image is cryptographically verified, integrity is guaranteed
Detection and blocking of image substitution	0/1 — attack not detected; a malicious image under the latest tag was deployed as an update	1/1 — attack blocked; signature verification failed, update was cancelled
Average incident response time	High: ≈2 hours (incident detected post factum, manually, through log analysis)	Low: ≈5 seconds (immediate detection during deployment, automatic blocking)
Protection against rollback attacks	0/1 — absent; an old image with a backdoor was redeployed as an "update"	1/1 — blocked; artifact rejected by freshness policy
Number of unplanned rollbacks / failures	2 (1 due to compromise, 1 due to deployment of an incorrect image)	0 — incidents neutralized prior to deployment
Version traceability and reliability	Low: the active service version is implicit (latest tag may refer to different builds), traceability is limited	High: each deployed image is uniquely identified by a digest hash and linked to a specific commit
Update process overhead	0% (baseline deployment duration ≈5 min)	≈+5% (≈15 s for signing and ≈10 s for verification) — negligible pipeline overhead
Trust level in the deployed image	Subjective: dependent on trust in the environment and processes	Objectively high: authenticity and integrity of each image are cryptographically verified

Unlike perimeter-based or purely organizational security mechanisms, artifact signing and mandatory verification provide infrastructure-independent proof of the authenticity and integrity of each deployment. The use of open OCI standards and Sigstore enables the solution to be scaled across heterogeneous environments without coupling to a specific cloud provider or orchestrator, while preserving transparency, reproducibility, and auditability of updates. According to the experimental evaluation, the achieved security improvements are accompanied by only minor operational overhead, making the approach suitable for practical adoption in modern DevSecOps-oriented processes. In this context, decision support systems for IT project management automation [35] can further facilitate the structured integration of security controls into agile development workflows.

5. Conclusions

This paper addresses the problem of secure container updates in server environments operating under automated CI/CD processes, where traditional deployment approaches do not provide guarantees of container artifact authenticity and integrity. It is substantiated that the compromise of any element in the software supply chain, including the container registry, delivery network, or CI/CD configurations, may lead to container image substitution and execution of untrusted code in production environments.

This work presents and implements a secure container update model that combines cryptographic signing of container images using Cosign in keyless mode (OIDC/Fulcio), mandatory automated signature verification at the deployment stage, immutable image addressing based on

SHA-256 digests instead of tags, and version and freshness control policies to mitigate rollback attacks while supporting controlled rollback exclusively to previously verified artifacts. The proposed approach is integrated without modifying the core CI/CD pipeline logic by introducing additional signing and verification stages, which ensures its practical applicability within DevSecOps processes.

Experimental modeling of attack and failure scenarios confirms the effectiveness of the proposed solution. The results demonstrate that the overhead introduced by the additional security stages is minimal and does not significantly affect the overall deployment time. Overall, the study shows that the adoption of container image signing using Cosign/Sigstore, mandatory pre-deployment verification, and digest-based image addressing substantially enhances the security of automated container updates in CI/CD environments, providing cryptographically grounded authenticity and integrity guarantees with minimal impact on delivery performance.

Declaration on Generative AI

During the preparation of this work, the authors used Grammarly in order to grammar and spell check, and improve the text readability. After using the tool, the authors reviewed and edited the content as needed to take full responsibility for the publication's content.

References

[1] Secure your container images with signature verification.

[2] State of the software supply chain report.

[3] D. Tymoshchuk, et al., An explainable artificial intelligence approach for detecting network attacks. CEUR Workshop Proc., 2025, 4141, pp. 38-51.

[4] D. Tymoshchuk, et al., Detection and classification of DDoS flooding attacks by machine learning method. CEUR Workshop Proc., 2024, 3842, pp. 184-195.

[5] S. Jena, A. Ranjan, V. Singh, DDoS Attack Detection Using Explainable AI in Machine Learning, In: Communications in Computer and Information Science, Springer Nat. Switz., Cham, 2025, pp. 3-16. doi:10.1007/978-3-031-83796-8_1.

[6] M. Stetsiuk, Y. Klots, D. Tymoshchuk, M. Karpinski, N. Petliak, Detection of multi-vector attacks in IoT networks: a graph attention network-based approach. CEUR Workshop Proc., 2025, 4146, pp. 296-313.

[7] Y. Klots, V. Titova, N. Petliak, D. Tymoshchuk, N. Zagorodna, Intelligent data monitoring anomaly detection system based on statistical and machine learning approaches. CEUR Workshop Proc., 2025, 4042, pp. 80-89.

[8] N. Petliak, Y. Klots, M. Karpinski, V. Titova, D. Tymoshchuk, Hybrid system for detecting abnormal traffic in IoT. CEUR Workshop Proc., 2025, 4057, pp. 21-36.

[9] D. Shevchuk, O. Harasymchuk, A. Partyka, N. Korshun, Designing Secured Services for Authentication, Authorization, and Accounting of Users. CEUR Workshop Proc., 2023, 3550. pp. 217-225.

[10] V. Chornii, Y. Martseniuk, A. Partyka, O. Harasymchuk, Information security risks associated with the uncontrolled storage of secrets in source code. CEUR Workshop Proc, 2025, vol. 4042 : Proc. cyber secur. data prot., CSDP 2025, Lviv, Ukraine, July 31, 2025, p. 250-271.

[11] Docker Inc, Home.

[12] SolarWinds supply chain attack.

[13] C. Okafor, T. R. Schorlemmer, S. Torres-Arias, J. C. Davis, SoK: analysis of software supply chain security by establishing secure design properties, In: CCS '22: 2022 ACM SIGSAC conference on computer and communications security, ACM, New York, NY, USA, 2022. doi:10.1145/3560835.3564556.

[14] I. Koishybayev, et al., Characterizing the security of github CI workflows. In: 31st USENIX Security Symposium (USENIX Security 22), pp. 2747-2763, 2022.

[15] GitHub Actions documentation — GitHub Docs.
[16] Docker Inc, Docker hub.
[17] P. Liu, et al., Understanding the security risks of docker hub, In: Computer security – ESORICS 2020, Springer Int. Publ., Cham, 2020, pp. 257-276. doi:10.1007/978-3-030-58951-6_13.
[18] Denial of Service (DoS) guidance.
[19] A. Mills, J. White, P. Legg, Longitudinal risk-based security assessment of Docker software container images, Comput. & Secur, 2023, 103478. doi:10.1016/j.cose.2023.103478.
[20] F. Wei, S. Tate, M. Ramkumar, S. Mohanty, A scalable trustworthy infrastructure for collaborative container repositories, Distrib. Ledger Technol. 1.1, 2022, 1-29. doi:10.1145/3554760.
[21] Enforcing image trust on Docker containers using Notary.
[22] The Update Framework Authors. (n.d.). *The Update Framework (TUF): A framework for securing software update systems*.
[23] Z. Newman, J. Meyers, S. Torres-Arias, Sigstore: Software signing for everybody, In: CCS '22: 2022 ACM SIGSAC conference on computer and communications security, ACM, New York, NY, USA, 2022. doi:10.1145/3548606.3560596.
[24] Kubernetes Authors. (n.d.). *Kubernetes documentation*. Kubernetes.
[25] GitHub — googlecontainertools/distroless: language focused docker images, minus the operating system.
[26] Eastlake, D., & Hansen, T. (2006). *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF) (RFC 6234)*. Internet Engineering Task Force (IETF).
[27] Sigstore Authors. (n.d.). *Cosign: Container signing, verification and storage in an OCI registry*. GitHub.
[28] How OpenID connect works — Openid foundation.
[29] Eastlake, D., & Hansen, T. (2011). *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF) (RFC 6234)*. Internet Engineering Task Force.
[30] R. Amir, Open-Source container security: A deep dive into trivy, clair, and grype, 2024.
[31] Nemorize. (n.d.). *Hermetic builds*.
[32] Y. Zhuravchak, D. Zhuravchak, A. Zhuravchak, I. Opirskyy, Security regression visualization in CI/CD using Trivy and GitHub Actions, CEUR Workshop Proc., 4042, 2025, 272-281.
[33] P. Vorobets, A. Horpenyuk, I. Opirskyy, Preparing IT Infrastructure for the Quantum Age: Post-Quantum SSH, CEUR Workshop Proc., 4016, 2025, p. 66-75.
[34] V. Vysotska, D. Baranovskyi, O. Lozynska, Y. Zhuravchak, Decision Support System Development for IT Project Management Automation to Support the LEAN Philosophy, Lect. Notes Mech. Eng., 2026, 37-66. doi:10.1007/978-3-032-10111-2_4.