Canary in the Container: Deception-Based Monitoring for Private Container Images

Billy Tsouvalas
Stony Brook University
vtsouvalas@cs.stonybrook.edu

Nick Nikiforakis
Stony Brook University
nick@cs.stonybrook.edu

***Abstract*—Private container images hold sensitive infrastructure configurations, production credentials, and internal service logic. When these images are accessed without authorization, whether through insider threats, credential compromise, or accidental leakage to public registries, organizations typically have no way to detect the breach. In this paper, we introduce AMONDIN, an automatic and application-agnostic framework, which instruments Docker images with two types of deception-based monitoring components: (i) static honeytokens, in the form of decoy API keys that detect secret extraction from image layers, and (ii) dynamic lifecycle beacons, i.e., HTTP callbacks that fire during the build, run, and exec stages of an image. AMONDIN integrates into CI/CD pipelines, allowing developers to monitor who accesses, extracts secrets, or interacts with their container images, focusing on private registries hosting sensitive infrastructure. We evaluate the monitoring capabilities of AMONDIN in an IRB-approved study, where we simulate the inadvertent exposure of private images on a public container registry, by deploying 100 instrumented images on DockerHub for 1 month. AMONDIN tracked the exfiltration of embedded decoy API keys and captured the subsequent exploitation attempts: over 60,000 AWS API requests from more than 3,000 unique IP addresses, and attackers exhausting all credits on our decoy Claude API keys.**

***Index Terms*—Deception, Containers, Honeytokens, Beacons**

I. Introduction

In the past few decades, software has been progressively evolving from dedicated, isolated scripts running on a specific machine towards a multi-faceted digital ecosystem characterized by layers of interconnected software, APIs, and databases, constantly communicating and processing data. Historically, deploying such complex software included tightly integrating it with a physical server.

Containerized applications constitute an alternative to that traditional way of deploying software, by packaging all necessary dependencies (code, system tools, libraries, configuration files) inside a self-contained portable environment. Compared to traditional host-based or Virtual Machine (VM) deployments, containerized applications offer resource efficiency, portability, consistency, fast deployment, as well as isolation/separation guarantees. Docker accounts for an estimated 86.83% of the containerization field [1], and its accompanying public registry, DockerHub, has become the de facto venue for containerized software, with a public catalog of over 14 million images and more than 11 billion pulls per month [2].

Given their popularity, Docker images are prime targets for attackers. Attack vectors against containerized applications include attacks inherited from the traditional host-based systems, such as arbitrary code execution, privilege escalation, bypassing authentication, supply chain attacks, and Denial of Service (DoS) [3], [4], as well as attacks exploiting the intrinsic characteristics of container images. Such container-specific attack vectors include targeting unpatched software vulnerabilities on containers [4], [5], spreading malicious images at scale [6], and even targeted escape exploits that break the isolation between container and host [7].

Among these, the most directly lucrative attack vector is the scanning of public registries for exposed secrets. Approximately 8.5% of Docker images on the publicly accessible DockerHub contain some type of secret (private keys, API keys, etc.) [8], while large-scale audits of public container registries have revealed hundreds of thousands of valid hardcoded credentials embedded inside publicly-available images [9], [10].

While these studies focus on public images, the implications for private images are even worse. Organizations build private, proprietary images containing internal service credentials with elevated privileges, infrastructure configurations, and operational information. Although such images are intended to remain private, i.e., hosted on access-controlled repositories of public registries, or private registries altogether, they may become exposed through CI/CD misconfigurations, insider threats, stolen registry credentials, or accidental publication. Once exposed, there is no built-in mechanism to detect that an image's secrets have been extracted.

The consequences of such exposure are well-documented. In 2021, an attacker managed to extract the credentials from a Codecov Docker image [11]. Using this credential, the attacker modified the company's uploader script and managed to silently exfiltrate customer credentials, tokens, API keys, and anything stored as environment variables from customers' CI/CD pipelines for two months, breaching hundreds of customer networks [12], [13]. Specifically regarding private images, a 2023 study by Trend Micro found over 20,000 images hosted on publicly exposed container registries that were accessible without authentication [14]. Among the owners of these images were "private organizations with no other public container or code-sharing projects", suggesting that these images were inadvertently made publicly accessible. The affected organizations had no way to detect that their images were being accessed and their secrets had been leaked.

These incidents highlight a fundamental gap: once a private image has been pulled or leaked, there is no mechanism to detect whether an attacker extracts its secrets, runs it in an unauthorized environment, or redistributes it.

To this end, we introduce AMONDIN, a deception-based defense and monitoring framework for private container images. AMONDIN seamlessly integrates into CI/CD pipelines and instruments Docker images with two types of monitoring elements: (i) *static honeytokens*, decoy API keys embedded in image layers that detect secret extraction, and (ii) *dynamic lifecycle beacons*, HTTP callbacks that fire when the image is built, run, or accessed via interactive shell. For images where the original Dockerfile is not available (e.g., third-party base images, legacy deployments), AMONDIN can also reconstruct a faithful Dockerfile from a published image and instrument it with its monitoring elements. The resulting image behaves identically to the original, but also covertly monitors for unauthorized access. This paper makes the following original contributions:

- We introduce AMONDIN, a CI/CD-integrated framework that instruments private container images with static honeytokens and dynamic lifecycle beacons, enabling image owners to detect unauthorized access, secret extraction, and image execution, whether the image is hosted on a private registry or has been inadvertently exposed on a public one.
- We demonstrate that AMONDIN successfully instruments 100 official Docker images, preserving the original image's functional and behavioral fidelity, while introducing no measurable overhead.
- During an IRB-approved, in-the-wild evaluation, AMONDIN detected secret leakage and surfaced every adversarial interaction that followed, from over 60,000 requests to our AWS honeytokens, to the complete exhaustion of our Claude API credits, demonstrating its capability to comprehensively monitor post-leakage attacker behavior.

II. Background and Motivation

A. Docker Overview

Docker is a framework that automates the deployment of lightweight, portable, containerized applications, called containers. These containers are created from Docker images, which are immutable, read-only, pre-packaged, layered units containing everything needed to run the application, e.g., code, runtime, system tools, libraries. Docker images are constructed using the `docker build` process, during which the Docker engine consumes a Dockerfile together with the associated build context (source files and directories). A Dockerfile is simply the recipe of commands and configuration directives necessary to build a given image from source. During the build, the engine processes each instruction sequentially and commits its result as a read-only image layer. To ensure consistency, portability, and efficiency, the build process typically includes a foundational base image (e.g., Alpine or Debian), rather than building the entire image from scratch. Alternatively to building custom local images, the `docker pull` command can fetch ready-made images from DockerHub.

DockerHub hosts over 14 million images, spanning all software categories. Images are classified as *official*, which are maintained by the software's original vendor, and *community* images, which may be published by any registered DockerHub user. Given the immense diversity and range of different offerings of the images hosted, there is no single image archetype. For instance, the Nginx official image runs a persistent daemon that listens on a specific port, whereas the Python image drops into an interactive Read-Eval-Print Loop (REPL) and exits immediately. Similarly, the PostgreSQL image deploys an entrypoint that initializes a data directory, and the Alpine image provides nothing more than a minimal filesystem with a shell. This diversity also extends to how images configure their runtime, via multi-stage builds, entrypoints, environment variables, volumes, etc.

After building or pulling an image, the `docker run` command instantiates a Docker image into a live container, creating an isolated runtime environment with specific resource constraints and network policies. Although containers are designed to be isolated and self-contained, oftentimes runtime access and observability are required. The `docker exec` command facilitates this by allowing shell access to the live container. This command is important since it breaks the live container's isolation boundary, albeit in a controlled manner, allowing access to the live application's configuration and data.

B. Protecting against attacks

To address the problem of malicious actors scanning at scale for vulnerabilities, Docker has attempted to ameliorate the situation by offering the Docker-native Scout service [15]. Docker Scout identifies the vulnerabilities of a given image by analyzing its layers and generates a Software Bill of Materials (SBOM), which it cross-references with known CVEs.

More recently, Docker introduced Hardened Images, which are security-first images that are continuously maintained and meet security and compliance requirements [16]. Although both of these solutions provide developers with useful tools to identify vulnerabilities and minimize the attack surface of their images, their protection is by no means comprehensive. Regarding secret leakage, the responsibility still lies with developers to adopt secret-mounting tools [17] and third-party scanners like TruffleHog [18] and GitGuardian [19]; measures that are voluntary and not widely adopted.

A common thread across the aforementioned protections is that they primarily focus on public images. Vulnerability scanners, hardened bases, and pre-publication secret-detection tools all assume a model where the image is intended for distribution. For images on private registries, or private images on restricted public repositories, the security picture is different. Private registries provide access-control mechanisms (IAM policies, token-based auth, role-based access controls) to restrict who can push and pull images. However, both registry controls and secret hygiene have repeatedly failed in practice.

In 2019, DockerHub itself suffered a database breach that exposed OAuth tokens for approximately 190,000 accounts [20]. Considering that these tokens granted read and

write access to private source code repositories linked to DockerHub, attackers managed to fully bypass the access-control mechanism. In 2026, attackers used a stolen push token to overwrite the security scanner Trivy on DockerHub with backdoored images, turning a trusted security framework into a credential-stealing tool [21].

These incidents demonstrate that once a private image gets leaked, whether by an unauthorized user, a compromised pipeline, an attacker who has bypassed registry controls, or through an unintentional publication by the owners, there is no built-in way to detect what happens next. It may be inspected for embedded secrets, executed in unauthorized environments, or redistributed to third parties. This leaves organizations blind to post-leakage threats against their most sensitive container assets.

C. Research Questions

AMONDIN closes this gap, instrumenting images with deception-based probes that detect unauthorized access, secret extraction, and execution regardless of registry. Our research questions are:

RQ1: Can AMONDIN automatically instrument arbitrary Docker images? Given the diversity of software in the container ecosystem, we evaluate whether AMONDIN can insert honeytokens and lifecycle beacons into arbitrary images without human intervention and without breaking functionality.

RQ2: Does the instrumentation preserve functional correctness and behavioral fidelity? We evaluate whether instrumented images behave identically to their originals in terms of runtime behavior, network configuration, startup time, and resource usage.

RQ3: Do the embedded honeytokens and lifecycle beacons detect real-world attacker interaction? We simulate the scenario where private images become publicly exposed, by deploying instrumented images on a public registry and evaluating whether the honeytokens and beacons detect malicious activity.

III. DESIGN

AMONDIN is an automated, image-agnostic framework that embeds two complementary deception probes into container images: *static honeytokens*, decoy API keys whose use reveals unauthorized secret extraction, and *dynamic lifecycle beacons*, HTTP callbacks at the build, run, and exec boundaries that signal unauthorized construction, instantiation, or shell access. Together they cover the full range of attacker interaction, from passive layer inspection to active execution.

A. CI/CD Pipeline Integration

In its primary mode, AMONDIN operates as a CI/CD pipeline step: the developer supplies the Dockerfile and build context, and AMONDIN appends lifecycle beacons to the appropriate instructions and copies the honeytoken files into the image before it is built and pushed as usual. This requires no manual effort beyond adding AMONDIN to the pipeline.

B. Image Reconstruction

When the original Dockerfile is unavailable (e.g., third-party base images, legacy deployments, or auditing published artifacts), AMONDIN instead reconstructs a functional, faithful Dockerfile directly from the published image and then instruments it as above. We detail this reconstruction pipeline in Section IV.

C. Dynamic Instrumentation: Lifecycle Beacons

AMONDIN's dynamic instrumentation consists of HTTP GET callback beacons injected at three image lifecycle boundaries. At each lifecycle stage, a beacon makes a request to a unique path on a monitoring server, identifying the image and the lifecycle event:

- **Build beacon:** Fires during `docker build`, detecting attempts to reconstruct the image from source.
- **Run beacon:** Fires during `docker run`, detecting unauthorized instantiation of the image as a live container.
- **Exec beacon:** Fires during `docker exec`, detecting interactive shell access into a running container.

D. Static Instrumentation: Honeytokens

In addition to lifecycle beacons, AMONDIN plants static honeytokens in each instrumented image. These are decoy API key files (e.g., `api_keys.json`) containing credentials for monitored web services such as AWS, OpenAI, and Claude. Honeytokens are embedded into image layers via a `COPY` instruction in the Dockerfile. Unlike beacons, which only fire during dynamic interaction with the image, honeytokens detect a distinctly different attacker behavior: *static secret extraction*. An attacker who unpacks layers, inspects the filesystem, or runs automated secret-scanning will discover the decoy keys and likely attempt to use them.

Honeytoken activations are monitored through an Event Listener that tracks API service logs (e.g., AWS CloudTrail, OpenAI usage dashboard, Anthropic API logs). We classify the usage of the decoy API keys as follows:

- **Validation:** A simple invocation of the credential against its intended service that tells the attacker whether the key is live, indicating that the secret has been extracted and is being checked for validity and viability.
- **Enumeration:** Requests that probe the credential's scope, services, and permissions, indicating reconnaissance vectors.
- **Exploitation:** Leveraging the credential for resource access or credit exhaustion (e.g., launching compute instances, credit consumption), indicating active abuse.

This three-level classification helps image owners assess the severity of the compromise and the attacker's intent. Moreover, the combination of static honeytokens and dynamic lifecycle beacons means AMONDIN detects attacker interaction whether or not the attacker actually runs the image. Together they form two complementary detection paths: a *dynamic path* triggered by an adversary interacting with an image and a *static path* triggered by offline analysis of the image's embedded content.

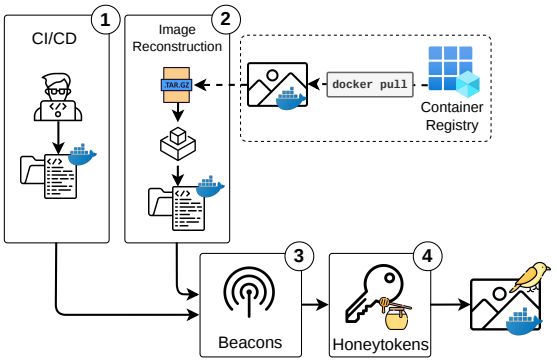


Fig. 1. AMONDIN instrumentation pipeline for CI/CD and image reconstruction

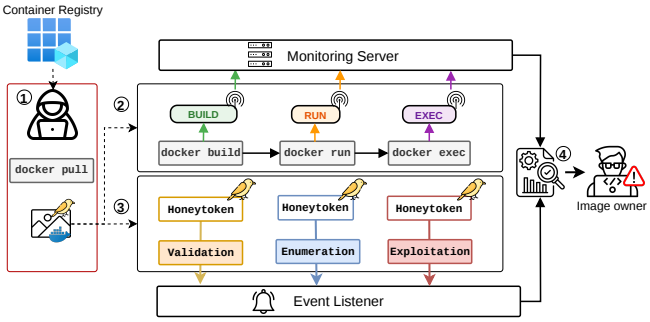


Fig. 2. Post-deployment monitoring architecture: honeytoken and beacon activation

E. Threat Model

We consider an organization whose private container images are exposed to unauthorized parties, through a registry breach, credential theft, misconfiguration, or accidental public exposure of an image, e.g., the inadvertent publication of a private image on a public container registry. An attacker who obtains the image may perform (i) *static extraction*, inspecting layers to harvest secrets and proprietary code, or (ii) *dynamic extraction*, building or running the image to probe its runtime behavior.

AMONDIN addresses both vectors: static honeytokens detect unauthorized layer inspection, while dynamic lifecycle beacons detect unauthorized execution and interaction.

IV. ARCHITECTURE

A. AMONDIN *instrumentation pipeline*

1) CI/CD mode: In Fig. 1, we present the AMONDIN instrumentation pipeline. Specifically, in CI/CD mode, the developer supplies AMONDIN with a Dockerfile and its associated build context (files, directories, etc.) directly. This mode is intended for image owners who want to instrument their own images as part of a build pipeline. In this case, AMONDIN skips the reconstruction stage entirely and forwards the Dockerfile to the *Beacon & Honeytoken Insertion* stage (③ - ④). This mode is recommended when the original build context is available.

2) Image reconstruction: When the original Dockerfile is unavailable (third-party images, auditing published artifacts, or instrumenting images from upstream vendors), AMONDIN reconstructs a functional Dockerfile directly from a pulled image. The reconstruction proceeds in three stages.

a) Dockerfile reconstruction: AMONDIN pulls the target image, runs `docker save` to obtain its tarball, and extracts the image configuration, metadata, and nested layer tarballs. It then invokes dfimage [22] to produce an approximate Dockerfile from the layer history. This approximation is far from functional: it carries syntax errors, hashes, shell-form commands, and no base image.

b) Base image detection: Most images use a base image to ensure portability, consistency, and to avoid creating an entire base OS from scratch. However, `FROM` instructions, which are how a base image is introduced into a Dockerfile, are not stored in the image's metadata. This is a problem because the `docker image history` command, which dfimage relies on, records the commands of each layer using that metadata. Since `FROM` instructions are not in the metadata, dfimage cannot determine which base image was used. To overcome this, AMONDIN extracts the first layer of the image and reads the `/etc/os-release` to identify the base image (e.g. `alpine:3.23`). Having located the base image, AMONDIN employs the `FROM` command, which corresponds to the original image's syntax. To ensure that the base image can always be located, we note that AMONDIN implements fallbacks in the base image search, i.e., image release files and configuration history.

c) Layer extraction: Even with the base image resolved, the approximate Dockerfile is not buildable. Image metadata stores commands in their shell-executed form, so bare `/bin/sh -c <command>` lines must be converted to proper `RUN` instructions. `COPY` instructions pose a bigger problem: they reference source paths from the original build context, which no longer exist. To recover the copied content, AMONDIN unpacks each layer tarball from the image into a local directory (one per layer), then rewrites every `COPY` instruction to read from the corresponding layer directory (`COPY <local_layer_dir> <same_destination>`). This restores each layer's original filesystem content and yields a functional Dockerfile that will successfully rebuild the published image from source.

3) Lifecycle beacon insertion: Once the Dockerfile and the relevant source files are available, either from CI/CD mode (①) or image reconstruction (②), AMONDIN injects HTTP callback beacons for the three lifecycle events: `build`, `run`, and `exec`. To ensure the beacons can fire, AMONDIN detects whether the Dockerfile installs `curl` or `wget` and uses whichever binary is present; if neither is available, it covertly appends the installation of one of them to an existing layer (alongside the installation of another binary).

The three beacons fire from different points. The `build`

beacon is appended into an existing layer of the reconstructed Dockerfile (so as not to increase the layer count) and fires whenever the image is built. Any attempt to rebuild the image from source will trigger a callback. The `run` beacon is placed inside the container's entrypoint script and fires when the image is run, or when a live container is created. The `exec` beacon overloads the shell binaries (`/bin/sh`, `/bin/bash`, etc.) with wrappers, firing whenever an interactive shell is spawned.

All three beacons are outbound HTTP requests requiring no listening ports and no runtime dependencies beyond `curl` or `wget`. Their placement preserves the image's original functionality while remaining invisible to casual inspection.

4) Honeytoken insertion: Lifecycle beacons detect when an image is built, run, or shelled into, but they do not detect *static* extraction of the image's embedded content. This is the case where an attacker inspects image layers to harvest secrets without ever running the image. To close this gap, AMONDIN embeds a decoy credentials file in each instrumented image, through one additional `COPY` instruction in the Dockerfile. The decoy credentials are tied to a monitored event listener, so any attempt to *use* them produces an alert that the image's layers have been inspected. Together, the lifecycle beacons (③) and honeytokens (④) cover both dynamic and static extraction vectors, giving image owners complete visibility into how their published images are accessed and used.

B. Honeytoken and beacon activation

In Fig. 2, we present AMONDIN's post-deployment monitoring architecture. We consider the scenario where an instrumented private image has been inadvertently exposed, e.g., through a registry breach, a stolen push/pull token, some misconfiguration, or accidental publication to a public repository, where an unauthorized party obtains a copy of the image (①). From AMONDIN's perspective, the attacker is indistinguishable from any other client who performs a `docker pull`, but the moment they begin to inspect the image, two independent detection paths activate.

The first detection path is the *beacon activation* (②). If the attacker attempts to rebuild the image from source, run it as a container, or obtain an interactive shell, the corresponding `build`, `run`, and `exec` beacons fire outbound HTTP callbacks to a Monitoring Server, identifying unauthorized dynamic interaction with the image.

The second detection path is the *honeytoken activation* (③). The decoy API credentials embedded in the image during instrumentation are tied to an Event Listener that observes any interaction with the honeytoken. Depending on the credential type and the provider's telemetry (e.g., AWS CloudTrail), these interactions can be classified into escalating levels of attacker engagement, e.g., merely *validating* the credential, *enumerating* its scope, permissions, and available services, or actively *exploiting* it to consume credits. Finally, both detection paths converge on a centralized analytics stage (④) that joins beacon callbacks with honeytoken events from the Event Listener. Rather than treating each signal in isolation, at this stage, we can reconstruct a coherent narrative of how an exposed image is being handled, e.g., whether attackers are simply inspecting its layers for secrets, rebuilding it locally to study its contents, running it to observe its behavior, etc. Since beacons cover the dynamic extraction vector (build, run, exec) and honeytokens cover the static extraction vector (layer inspection), correlating them gives image owners a holistic view of attacker behavior across the full lifecycle of an image. This means that image owners can have a full monitoring oversight from the initial pull, through the inspection, and all the way to the attempted abuse.

V. SYSTEM EVALUATION

We evaluate AMONDIN, focusing on two aspects of the resulting instrumented images: their *functional correctness* and their *behavioral fidelity*, with respect to the original images. We specifically evaluate AMONDIN's *image reconstruction mode*, where the original Dockerfile is unavailable and must be rebuilt from the published image. The image reconstruction mode is the harder of the two input modes, since the CI/CD mode operates on a Dockerfile supplied by the developer and bypasses the reconstruction pipeline entirely. In terms of functional correctness, we evaluate the instrumented image in the following ways:

- Successful `build`: The image Dockerfile can be reconstructed from its extracted layers and built into a valid image, while the injected beacons do not break the build procedure of the image.
- Successful `run`: The container's entrypoint and startup sequence execute correctly; the reconstructed image preserves the runtime behavior of the original.
- Successful interactive shell into live container (`exec`): A user can interactively shell into the running container; the shell environment, user permissions, and filesystem state are intact.
- Correct beacon behavior: The instrumented image fires exactly the beacons that are expected for its category, i.e., daemon/service images fire all three (`build`, `run`, `exec`); REPL, build tool, and base OS images exit immediately after getting instantiated.

In terms of behavioral fidelity of the instrumented images, we evaluate whether the instrumented image behaves the same as the original. In order to verify that AMONDIN does not alter the functionality of the original image, we focus on the following components:

- Network port comparison: The instrumented and the original image listen on the same ports at runtime, validating that the injected beacons do not interfere with the application's network binding.
- Startup time comparison: We measure the startup time overhead introduced by the beacon instrumentation. A negligible difference in startup times confirms that the beacons do not degrade the container's responsiveness.
- Resource usage comparison (CPU, Memory): We evaluate whether the instrumented container consumes comparable

TABLE I
SUCCESSFULLY INSTRUMENTED DOCKER IMAGES BY IMAGE TYPE

Category	Count	Representative Examples
Daemon/Service	61	`nginx`, `postgres`, `wordpress`, `redis`, `mongo`, `phpmyadmin`, `joomla`
REPL/Interactive	24	`bash`, `node`, `ruby`, `golang`, `perl`, `php`, `haskell`
Build Tool	6	`maven`, `gradle`
Base OS	9	`ubuntu`, `debian`, `alpine`, `fedora`, `amazonlinux`
Total	**100**	

resources to the original. Comparable resource usage confirms that the injected beacon logic is lightweight.

A. Experimental Setup

To evaluate AMONDIN along the outline above, we employed official Docker images. Official Docker images span a wide range of software categories, including databases, web servers, programming languages, operating systems, and build tools, making them representative of real-world containerized workloads. They also include diverse packaging patterns (different base images, entrypoint scripts), which allows us to further evaluate AMONDIN's generalizability.

We evaluated 100 official Docker images, covering all major image categories, including daemons and services (e.g., `nginx`, `postgres`), interactive runtimes and languages (e.g., `php`, `node`), build tools (e.g., `maven`, `gradle`), and base operating systems (e.g., `ubuntu`, `alpine`), as described in Table I. For each of these images, AMONDIN pulled their latest image from DockerHub, reconstructed their Dockerfile and injected the lifecycle beacons and the static honeytokens (decoy AWS, OpenAI, and Claude API keys). We tested each instrumented image by building it, running the container, and executing a command inside it, verifying that the `build`, `run`, and `exec` beacons fire correctly at each stage.

Across all 100 images, AMONDIN performed this instrumentation without any manual intervention, answering **RQ1** in the affirmative.

B. Functional and Behavioral Fidelity

For all evaluated images, AMONDIN successfully reconstructed the Dockerfile and instrumented the image with both lifecycle beacons and static honeytokens. The expected lifecycle stages depend on the image category: daemon and service images (e.g., `nginx`, `postgres`) run a long-running foreground process and support all three stages (`build`, `run`, `exec`), whereas REPL, build-tool, and base-OS images (e.g., `php`, `maven`, `ubuntu`) exit after a single command unless invoked interactively. Under these category-specific constraints, all 100 instrumented images succeeded at their expected stages, with each beacon firing correctly, while the instrumented images remained functionally indistinguishable from their originals, with every service starting correctly and every endpoint returning the same response.

In Table II, we present the results of the behavioral fidelity results for AMONDIN. The instrumented images preserve

TABLE II
BEHAVIORAL FIDELITY METRICS

Metric	Value
Network port match	100%
Avg startup time delta	0.093s
Avg CPU delta	−0.032%
Avg memory delta	0.003%

network port behavior in 100% of cases, confirming that neither the beacons nor the honeytokens interfere with the applications' network bindings. The average startup time, CPU, and memory overheads are all negligible; on average, instrumented containers take only 0.093s longer to start, use 0.032% less CPU, and 0.003% more memory than their uninstrumented, original counterparts. In fact, the CPU delta is slightly *negative*, which is a clear indication that the instrumentation overhead is smaller than the natural run-to-run variability of the measurements themselves: the noise is greater than the difference. These results confirm that AMONDIN's instrumentation process preserves the functional behavior and runtime characteristics of the original image.

Regarding **RQ2**, we conclude that the AMONDIN instrumentation preserves functional correctness and runtime behavior, while it introduces no measurable overhead.

VI. IN-THE-WILD EVALUATION

To validate AMONDIN's monitoring capabilities against real-world adversaries, we simulate the inadvertent publication of a private image on a public registry. To that end, we deploy 100 AMONDIN-instrumented nginx images on DockerHub for one month. To accurately simulate an accidental leak, the images are not advertised and bear generic names suggestive of web technologies (e.g., `query-parser-service`). This experimental setup lets us evaluate in a real-world scenario whether AMONDIN's honeytokens and beacons successfully detect unauthorized access to an exposed image.

A. Ethical Considerations

Our user study protocol was reviewed by our IRB before any data collection began. The approval covered the deployment of the instrumented images on DockerHub. The deployed images were not advertised anywhere and did not pose any adverse risk to legitimate users that may interact with them. All decoy API keys were provisioned under accounts with arbitrary funds, strict safeguards and, wherever possible, hard spending caps were set.

B. Honeytoken activations

Over one month, the simulated leaked images accumulated 31,165 pulls. AMONDIN captured the extraction and use of every one of the 100 decoy AWS keys. Overall, our AWS honeytokens registered 61,584 API calls spanning validation, enumeration, and exploitation attempts against our AWS resources, and originating from 3,020 unique IP addresses. The OpenAI and Claude honeytokens both registered steady, low-volume validation calls from the outset, and after approximately three weeks, the Claude honeytokens captured a single

high-volume burst of requests that drained the account balance entirely. AMONDIN's monitoring tracked every abuse of the embedded credentials, demonstrating that the honeytokens are a reliable signal of secret exfiltration and exploitation from an inadvertently exposed image.

C. Beacon activity

Over our month-long simulation of private image leaks, none of our deployed beacons fired. The absence of beacon activations is itself a key finding. The actors interacting with our images did not build, run, or shell into any of the 100 deployed images. Combined with the honeytoken abuse as described above, this suggests that adversaries employed static analysis: unpacking image layers without running the container. AMONDIN's static and dynamic detection paths render this distinction between attacker workflows observable to the image owner.

Regarding **RQ3**, our simulation of leaked private images demonstrates that **AMONDIN's honeytokens and beacons detect real-world attackers**, turning post-leak activity into an observable signal for image owners.

VII. Discussion

Honeytokens and beacons detect disjoint adversary behaviors: static secret extraction and dynamic build/run/exec interaction. In our deployment the honeytokens saw heavy abuse while no beacon ever fired, suggesting static-only attack vectors. However, it is possible that attackers may instead have run the images in isolated environments that suppressed the outbound callbacks. Either way, the honeytokens still flag the compromise, since credential abuse surfaces at the cloud provider rather than via a callback from the attacker's host, demonstrating that AMONDIN's two detection paths are complementary, with coverage that holds even when one channel is evaded or suppressed.

Once a private image is exposed, the resources inside it can no longer be protected, and no defenses can be retrofitted after the fact. AMONDIN addresses this by instrumenting images at build time, as a CI/CD step alongside vulnerability scanning tools that owners already routinely run. This turns each image into its own active sensor, shifting post-leakage monitoring from a reactive investigation into a continuous defensive capability.

VIII. Related Work

Protecting containers. Prior work on securing container images has mainly focused on hardening the runtime and supply chain of images. This includes attack-surface reduction approaches employing debloating and syscall restrictions [23], enforcing stricter access control policies [24], and image-based vulnerability detection [25], [26]. However, these protections focus on the runtime and pre-deployment phases, leaving post-deployment monitoring and secret-leakage detection as comparatively underexplored directions.

On the other hand, containers have proven to be effective primitives for building honeypots and observing attacker behavior [27], with monitoring-adjacent efforts extending to the orchestration layer [28]. However, little attention has been paid to developing monitoring tools for the images themselves and their embedded contents. As such, in the present project, we introduce AMONDIN, an automatic framework that injects non-interfering static honeytokens and monitoring beacons into container images to monitor and protect them.

Deception and Beacons. Decoys are deceptive defense artifacts designed to be indistinguishable from real assets, luring attackers into interacting with them and thereby revealing their presence [29], [30]. Beacons extend this concept by embedding signals within these decoys, so that any unauthorized access or exfiltration is automatically detected when the beacon triggers, transforming passive deception into an active detection mechanism.

Building on this primitive, defenders embed hidden triggers that fire when an adversary performs a specific action across a wide range of digital assets. Decoy documents and filesystems have been deployed to detect insider threats and breaches [31], decoy hyperlinks have been employed to flag adversaries [32], and deceptive endpoints have been added to web applications to detect and deceive probing adversaries [33], [34]. More closely related to our work, prior efforts have planted decoy credentials in the wild to observe how attackers discover and abuse exposed secrets [35], [36]. AMONDIN applies this same action-triggered, decoy logic to monitor the container supply chain, injecting images with beacons and honeytokens that notify image owners of unauthorized accesses and exfiltration attempts against their containerized resources.

Secret Leakage. In 2023, an internet-wide analysis evaluating over 330,000 DockerHub images revealed that 8.5% of them contained secrets, exposing over 52,000 private keys and more than 3,000 API secrets [8]. More recently, Flare's scanning of DockerHub identified over 10,000 images containing live credentials, including nearly 4,000 access tokens for LLM services such as OpenAI and Anthropic [9]. Another large-scale study in 2025 revealed the existence of 100,000 valid secrets on images publicly available on DockerHub [10].

Despite efforts to tackle secret leakage, e.g., securing secrets during image construction, or eliminating them from the image altogether [18], [37], the problem persists at scale. Moreover, the risk is considerably greater when a *private* image, which typically contains credentials, proprietary code, and sensitive details, is inadvertently pushed to a public registry. AMONDIN fortifies private images with enhanced monitoring capabilities by embedding deceptive beacons and decoy credentials directly into the image. This allows image owners to detect leakage and track any unauthorized interactions that follow.

IX. Conclusion

In this paper, we introduced AMONDIN, an automatic, image-agnostic framework that instruments container images with deception-based monitoring probes. AMONDIN embeds within the image both static honeytokens (decoy API keys that detect secret extraction) and dynamic lifecycle beacons (HTTP callbacks that detect image interaction), while seamlessly integrating into CI/CD pipelines. This allows image

owners to detect unauthorized access to their private container images, whether those images reside on private registries or have been inadvertently exposed on public ones. We showed that AMONDIN successfully instruments 100 official Docker images across all major container categories, while preserving functional and behavioral fidelity. Last, we validated its monitoring capabilities by deploying 100 instrumented images on DockerHub, simulating the accidental publication of a private image, where AMONDIN captured substantial malicious activity, including over 60,000 event requests from more than 3,000 unique IP addresses on our AWS honeytokens and the complete exhaustion of our Claude API credits, validating it as an indispensable defensive asset for private image owners.

Acknowledgments: This work was supported by the Army Research Office (ARO) under grant W911NF-24-1-0051. We used Anthropic's Claude Opus 4.6 to assist with the development and debugging of AMONDIN, and for manuscript editing. All LLM outputs were manually reviewed and verified.

REFERENCES

[1] 6sense, "Docker Market Share," https://6sense.com/tech/containerization/docker-market-share, 2025, [Online; accessed 14-November-2025].

[2] Docker, "Docker Hub," https://www.docker.com/products/docker-hub/, 2025, [Online; accessed 14-November-2025].

[3] H. Shi, L. Ying, L. Chen, H. Duan, M. Liu, and Z. Xue, "Dr. docker: A large-scale security measurement of docker image ecosystem," in *Proceedings of the ACM on Web Conference*, 2025.

[4] M. S. Haq, T. D. Nguyen, A. Ş. Tosun, F. Vollmer, T. Korkmaz, and A.-R. Sadeghi, "Sok: A comprehensive analysis and evaluation of docker container attack and defense mechanisms," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 4573–4590.

[5] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on docker hub," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017.

[6] S. Pang, Y. Wang, Y. Yao, Z. Jiang, Z. Fan, and B. Liu, "Detection and analysis of poisoned image in container registry," in *2025 IEEE 36th ISSRE*. IEEE, 2025.

[7] M. Zhou, X. Jia, H. Su, S. Huang, Y. Du, H. Du, R. Wang, and J. Tang, "Container privilege escalation and escape detection method based on security-first architecture," in *IEEE HPCC/DSS/SmartCity/DependSys*. IEEE, 2023.

[8] M. Dahlmanns, C. Sander, R. Decker, and K. Wehrle, "Secrets revealed in container images: an internet-wide study on occurrence and impact," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, 2023.

[9] A. Morag, "Flare: Thousands of Exposed Secrets Found on Docker Hub, Putting Organizations at Risk," https://flare.io/learn/resources/docker-hub-secrets-exposed, 2025, [Online; accessed 16-February-2026].

[10] GitGuardian, "Fresh From The Docks: Uncovering 100,000 Valid Secrets in DockerHub," https://blog.gitguardian.com/fresh-from-the-docks-uncovering-100-000-valid-secrets-in-dockerhub/, 2025, [Online; accessed 15-February-2026].

[11] Codecov, "Codecov security update," https://about.codecov.io/security-update/, 2021, accessed 2026-04-08.

[12] A. Sharma, "Hundreds of networks reportedly hacked in codecov supply-chain attack," https://www.bleepingcomputer.com/news/security/hundreds-of-networks-reportedly-hacked-in-codecov-supply-chain-attack/, 2021, accessed 2026-04-08.

[13] HashiCorp Security, "HCSEC-2021-12: Codecov security event and HashiCorp GPG key exposure," https://discuss.hashicorp.com/t/hcsec-2021-12-codecov-security-event-and-hashicorp-gpg-key-exposure/23512, 2021, accessed 2026-04-08.

[14] Trend Micro, "Exposed container registries: A potential vector for supply chain attacks," https://www.trendmicro.com/vinfo/us/security/news/virtualization-and-cloud/exposed-container-registries-a-potential-vector-for-supply-chain-attacks, 2023, accessed 2026-04-08.

[15] Docker, "Docker Scout," https://docs.docker.com/scout/, 2025, [Online; accessed 16-February-2026].

[16] ——, "Docker Hardened Images," https://www.docker.com/products/hardened-images/, 2025, [Online; accessed 16-February-2026].

[17] ——, "Docker Secrets," https://docs.docker.com/engine/swarm/secrets/, 2025, [Online; accessed 16-February-2026].

[18] T. Security, "TruffleHog," https://github.com/trufflesecurity/trufflehog, 2025, [Online; accessed 16-February-2026].

[19] GitGuardian, "Integrate Docker Hub," https://docs.gitguardian.com/internal-monitoring/integrate-sources/container-registries-integrations/docker-hub, 2025, [Online; accessed 16-February-2026].

[20] L. Abrams, "Docker Hub database hack exposes sensitive data of 190k users," https://www.bleepingcomputer.com/news/security/docker-hub-database-hack-exposes-sensitive-data-of-190k-users/, 2019, accessed 2026-04-08.

[21] Docker, "Trivy supply chain compromise: What Docker Hub users should know," https://www.docker.com/blog/trivy-supply-chain-compromise-what-docker-hub-users-should-know/, 2026, accessed 2026-04-08.

[22] dfimage, "dfimage GitHub repository," https://github.com/LanikSJ/dfimage?tab=readme-ov-file, 2025, [Online; accessed 14-November-2025].

[23] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *RAID*, 2020.

[24] Y. Sun, D. Safford, M. Zohar, D. Pendarakis, Z. Gu, and T. Jaeger, "Security namespace: making linux security frameworks available to containers," in *USENIX Security 18*, 2018.

[25] S. Kwon and J.-H. Lee, "Divds: Docker image vulnerability diagnostic system," *Access*, 2020.

[26] S. H. Majumder, S. Jajodia, S. Majumdar, and M. S. Hossain, "Layered security analysis for container images: Expanding lightweight pre-deployment scanning," in *PST*. IEEE, 2023.

[27] V. D. Priya and S. S. Chakkaravarthy, "Containerized cloud-based honeypot deception for tracking attackers," *Scientific Reports*, vol. 13, no. 1, 2023.

[28] N. Spahn, N. Hanke, T. Holz, C. Kruegel, and G. Vigna, "Container orchestration honeypot: Observing attacks in the wild," in *RAID*, 2023.

[29] B. M. Bowen, P. Prabhu, V. P. Kemerlis, S. Sidiroglou, A. D. Keromytis, and S. J. Stolfo, "Botswindler: Tamper resistant injection of believable decoys in vm-based hosts for crimeware detection," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2010, pp. 118–137.

[30] B. M. Bowen, V. P. Kemerlis, P. Prabhu, A. D. Keromytis, and S. J. Stolfo, "Automating the injection of believable decoys to detect snooping," in *Proceedings of the third ACM conference on Wireless network security*, 2010.

[31] M. Ben Salem and S. J. Stolfo, "Decoy document deployment for effective masquerade attack detection," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 8th International Conference; DIMVA 2011, Amsterdam, The Netherlands, July 7-8, 2011. Proceedings 8*. Springer, 2011, pp. 35–54.

[32] D. Gavrilis, I. Chatzis, and E. Dermatas, "Flash crowd detection using decoy hyperlinks," in *IEEE International Conference on Networking, Sensing and Control*. IEEE, 2007.

[33] B. Tsouvalas and N. Nikiforakis, "Knocking on admin's door: Protecting critical web applications with deception," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2024.

[34] ——, "Defense of the clones: Securing web applications with automatic honeypot generation and deployment," in *2025 APWG Symposium on Electronic Crime Research (eCrime)*. IEEE, 2025, pp. 1–17.

[35] J. Onaolapo, E. Mariconti, and G. Stringhini, "What happens after you are pwnd: Understanding the use of leaked webmail credentials in the wild," in *Proceedings of the 2016 IMC*, 2016.

[36] S. Chakravarty, G. Portokalidis, M. Polychronakis, and A. D. Keromytis, "Detecting traffic snooping in tor using decoys," in *RAID 14th International Symposium*. Springer, 2011.

[37] Docker, "How Secrets Leak out of Docker Images," https://docs.docker.com/engine/swarm/secrets/, 2026, [Online; accessed 15-February-2026].