Uncontained Danger: Quantifying Remote Dependencies in Containerized Applications

Chris Tsoukaladelis
Stony Brook University

Roberto Perdisci
University of Georgia

Nick Nikiforakis
Stony Brook University

***Abstract*—Containers benefit software developers, aiding them with increased portability, scalability, and consistency across different environments. From a security point of view, containers incentivize the conversion of monolithic software into microservices which then can be isolated from each other, better lending themselves to least-privilege deployments.**

In this paper, we shed light to the unexplored issue of remote dependencies in Docker images and containers. Unless a Docker image is fully self-contained, every dependence to the outside world is an opportunity for attackers to hijack these dependencies and conduct supply-chain attacks against these images. To do so, we curate a dataset of 200K Docker images and design DockerGym, a dynamic analysis system which automatically installs, executes, and stimulates running containers, while monitoring their network communications. We discuss multiple approaches for activating the images in our dataset and the types of remote dependencies that we were able to discover. Among others, we observe that 13% of evaluated Docker images have some form of remote dependencies, with approximately 10K images resolving public domain names. We observe the use of unencrypted protocols (such as HTTP) and a range of other issues that could be straightforwardly exploited by attackers in the context of supply-chain attacks.

I. INTRODUCTION

Application containerization has helped solve multiple issues that engineers have faced over the years. Regarding consistency, developers now have the ability to package all their dependencies, as well as the application itself into a single unit that generally works across platforms, making the colloquial "works on my machine" largely a thing of the past. Efficiency and scalability – especially when compared to virtual machines – are also strong points for the use of containers. Moreover, containers are closely tied to the notion of microservices that allow developers to not only use the least-privilege principle more aggressively, but also roll out updates of their infrastructure in a continuous fashion.

While there exist multiple competing technologies and platforms for containerizing software, Docker is currently the most popular containerization technology, with tens of thousands of companies reporting that they are using it in different contexts [4], [5]. A particularly important part of the Docker ecosystem is the Docker Hub registry, a cloud-based marketplace for Docker images that is currently boasting over 15 million repositories containing images. Using Docker hub, developers can `docker pull` a large selection of ready-made images ranging from databases and web servers, to message queues, development environments, and machine-learning pipelines. These images can be used either off-the-shelf, or as base layers on top of which other functionality can be added and in turn be offered as a new Docker image.

To enable developers to use a specific version of a Docker image that is the most appropriate for their environment, Docker images typically carry tags. These tags can be used to denote compatibility with a specific operating system (e.g. `nginx:stable-bookworm` refers to the latest stable version of the nginx webserver for the Debian 12 distribution), the bundling of an image with additional software (e.g. `nginx:perl` indicating nginx with Perl support) and the referring to older, still-supported versions of an image (e.g. `nginx:1.25.4` as opposed to `nginx:latest`).

When using these specific tags (with perhaps an exception for the `latest` tag), there is an implicit expectation from the point of developers that these images will always perform the same operations and behave in the same manner. From a security point of view, someone who vets a Docker image carrying a specific tag today and concludes that it is benign, would expect that that image will behave near identically in the following weeks, months, and years. Moreover, the ability to address images via their digests (e.g. `nginx@sha256:f9c014f9ae6...`) is meant to provide further guarantees against image tampering, even when an image developer's account is compromised.

In this paper, we challenge this notion of image reproducibility by observing that Docker images (particularly the ones including full software pipelines as opposed to bare-bone software components) are not always self-contained. The software included in an image is free to reach out to the "live" web and fetch code and data when the image is deployed and while the software is running. In this way, even if the software contained in the Docker image does not change, the outside world *can* change and turn malicious between runs. As such, a Docker image that was benign last year can suddenly turn malicious if a remote server on which it depended was compromised or if a domain that it resolves expired and was re-registered by an attacker. The danger that arises from Docker Images contacting remote, unauthorized domains can not be understated. In February of 2025, `Safe{Wallet}`, a multi-signature platform for cryptocurrency, was hacked leading to a $1.5 billion USD crypto heist [11]. In their forensic investigation they reported that this was due to their Docker project communicating with a domain controlled by the attackers. [8] This type of threat is completely orthogonal to prior work by the community on the use of known vulnerable components in Docker images [28],

[40], container-escape attacks [1]–[3], [19], [26], [37], [45] and attacks at the level of the registry [27].

To this end, we conduct the first large-scale analysis of remote dependencies on the Docker Hub repository and the ability of attackers to perform supply-chain attacks using these dependencies. Namely, we develop a custom crawler to gather a dataset of 200K popular Docker images and build DOCKERGYM, a dynamic analysis system using multiple pipelines that install, execute, and stimulate these images in different ways, while monitoring their upstream network activity. Among other approaches, we explore the use of Large Language Models to automatically identify the necessary parameters for successfully launching each image by leveraging its public documentation. Additionally, we make use of network scanners and web crawlers to exercise more code paths in the launched containers. By analyzing the collected network data, we identify a wide range of insecure behaviors ranging from the use of unencrypted protocols, the reliance on expired domain names, and the extent to which supposedly self-contained Docker images depend on the Internet.

Our paper's primary contributions are as follows:

1) **Docker image dataset:** We built a custom Docker Hub crawler that allowed us to collect a dataset of 200K Docker images, which we then used for our experiments. As image discovery on Docker Hub is becoming harder and harder, we consider this an important contribution to researchers that want to follow up on our work.
2) **Docker execution and network capture system** To monitor the network traffic of 200K diverse Docker images, we designed a novel dynamic analysis system, DOCKERGYM, capable of automatically launching these images in a scalable and isolated fashion. To run the containers, we use two main approaches: a "vanilla" approach (e.g. using the same default options, regardless of image) and a "guided" approach making use of LLMs for interpreting image documentation from Docker Hub and extracting appropriate parameters needed to correctly launch an image. Similarly, we implemented multiple options of "stimulating" these Docker images, including interacting with the launched services via network scanning as well as by using web crawlers.
3) **Docker traffic analysis:** We discuss the network behavior of the Docker images in our dataset, comparing and contrasting the results obtained from the various launch and stimulation options. We then apply established network security practices to gauge their (in)security.

II. BACKGROUND

In this section, we provide a brief overview of how Docker images are constructed and how remote dependencies can be introduced during the build process and the execution of new containers from existing images. We also present the threat model that we consider in this paper.

```
1  FROM python:3.9
2
3  # Copy local app into container
4  WORKDIR /app/local
5  COPY app.py app.py
6
7  # Clone remote app into container
8  WORKDIR /app/remote
9  RUN git clone https://github.com/example/repo
10
11 EXPOSE 80 443
12
13 # Run app.py when the container launches
14 CMD ["python", "/app/local/app.py"]
15 CMD ["python", "/app/remote/app.py"]
```

Fig. 1: *Example Docker file with two different types of remote dependencies.*

```
1  def update_app(url):
2    print(f"Updating from {url}")
3    ...
4
5  def fetch_update():
6    response = requests.get("https://example.com
         /update")
7    data = response.json()
8    update_url = data.get("update_url")
9    update(update_url)
```

Fig. 2: *Local app.py embedded by the Dockerfile*

A. Dockerizing Applications and Remote Dependencies

Figure 1 shows a short Dockerfile used to "Dockerize" a Python application. Using the `python:3.9` image as its base layer, the new image includes a local application `app.py` (Line 5) and a remote application, cloned from *github.com* whenever a system executes a new Docker image (Line 9).

Beyond the implicit dependencies inherited by the base layer, this new Docker image contains two types of explicit dependencies. When the image is created, the local application `app.py` (shown in Figure 2) is copied into the image. Even though that file will always stay the same in all future containers created via this image, the copied code dynamically depends on `example.com` to check for app-specific updates. As we point out throughout this paper, there are no guarantees of who will be in control of `example.com` in the following weeks, months, and years *after* this image is created and how they will respond to any requests. Second, the remote GitHub repository is cloned whenever a new image is installed from this Dockerfile. As such, between image installations, new arbitrary code can be pushed to that repo, either by the owner of that repo or by attackers who were able to hijack it.

In both cases, any security analysis that aims to label this image as benign or malicious will not be able to account for future changes to `example.com` and `https://github.com/example/repo`. Even if users of this image refer to it via its digest (e.g. `docker pull example/example@sha256...`), the end software ex-

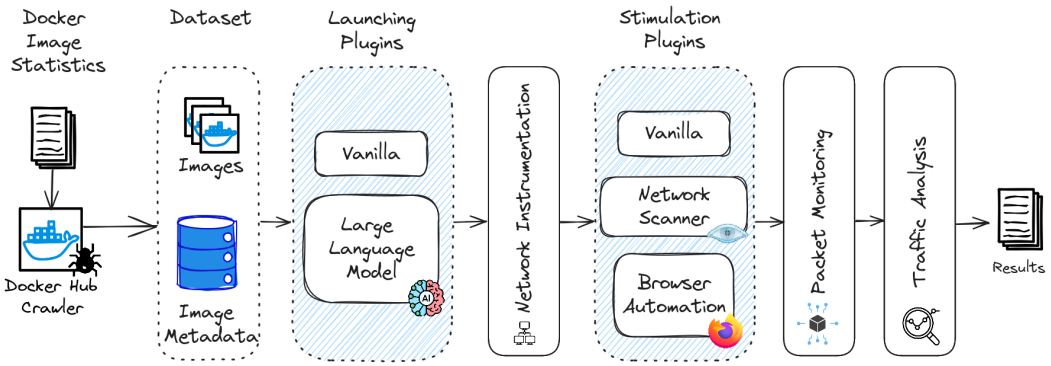

Fig. 3: *High-level architecture of* DOCKERGYM*, a dynamic-analysis system for evaluating the remote network dependencies of Docker images.*

ecuting on a system can be radically different than what developers anticipate.

B. Threat model

All Docker images that are not fully self-contained (i.e. they include one or more remote dependencies manifesting during their execution) provide opportunities for attackers to compromise them via supply-chain attacks. Attackers can abuse these remote dependencies either by obtaining access to the host being referenced (e.g. via an expired domain name) or, at the network level, by abusing insecure communications (e.g HTTP and other plaintext protocols) between a running Docker image and the outside world.

III. MEASUREMENT SYSTEM

In this section, we describe DOCKERGYM, the dynamic analysis system we designed and implemented to gather and analyze container-produced network data. Figure 3 shows a high-level overview of our system. The following paragraphs elaborate upon the individual parts of DOCKERGYM.

A. Docker Hub Crawler and Image Dataset

The first step in our process is to procure a large dataset of Docker images. To that effect, we turned to the largest public registry of containerized applications, Docker Hub. For the purposes of Docker Hub, we consider an image to be the combination of the repository and the image name, e.g. "ubuntu/nginx" for the "nginx" image name of the "ubuntu" repository. We also made the decision to use the `latest` tag of every image, to simplify our pipeline.

In terms of data composition, we aim to include a mix of the most popular Docker images, as well as a more generic sample of images with average and low popularity. To build our Docker Hub crawler we rely on Docker Hub's v2 search API, which we leverage in the following way:

First, we query the Docker Hub API for its most popular images (e.g. the images with the highest pull count) and save the resulting images in our dataset. Generally speaking, when querying the v2 API (which we found to be a better choice than the v3 API, when it comes to image discovery), a single search request will return up to 100 web pages listing up to 100 images each, so the maximum amount of unique images we could discover from a specific query is 10,000. In the case of most popular images, we used the v1 API of `store.docker.com`, which allows for sorting images based on their pull count.

After collecting these 10,000 most popular images, we look towards official, also known as "library", images [1], and their unofficial community alternatives. First, we procured a list of the official images by querying the Docker Hub API (via the `/v2/repositories/library` endpoint). We then queried the API again for community images sharing part of those names. The results of these queries include the official image, as the subject of the query and a list of all community-supplied images related to (or derived from) it, which we also collected.

Through this process, and after removing duplicate images that appeared in multiple search results, we were able to obtain a dataset of 200K images. Figure 4 shows the distribution of pull counts per image (as measured by Docker Hub). As we can see, the median pull count for the images in our dataset is about 100.

B. Launching Docker Containers

Due to the large size of our image dataset, we had to devise a scalable way for launching, collecting network data from, and stopping containers built based on these images. After a number of pilot experiments, we opted for executing multiple containers in parallel while enforcing network isolation, and for terminating the containers every hour, in order to free up the resources necessary to run the next batch of containers.

Similarly, given the size of the dataset, it would also be infeasible for us to manually determine the correct parameters necessary to run each and every Docker image. Therefore, to launch each image we use two different strategies: (i) running a container with a common set of default parameters, and (ii)

[1]In the context of Docker Hub, official images are a small set of images curated by Docker Hub and composed of popular Docker open source repositories. This is in contrast with Docker images published and maintained by community members or reputable open source entities, such as ubuntu

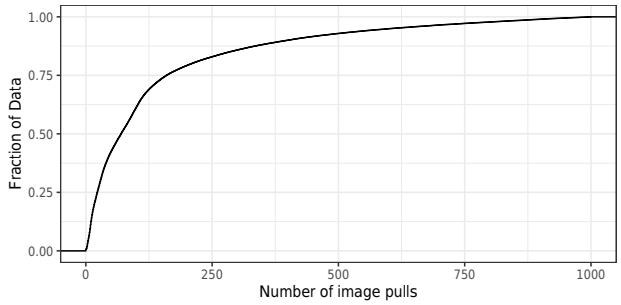

Fig. 4: Empirical Cumulative Distribution of Docker images in our dataset, according to their pull count. We have set the cut-off at 1,000 image pulls for visibility purposes, however there are images with millions of pulls.

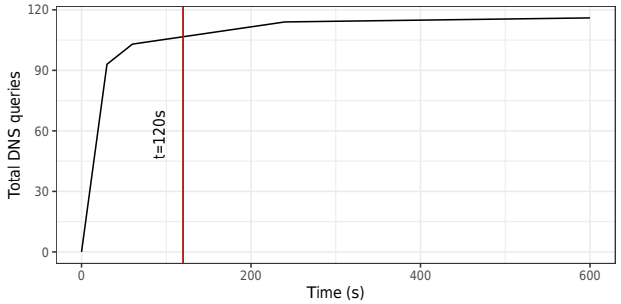

Fig. 5: Sampling of DNS queries over time. We selected to set our cut-off for network monitoring at 2 minutes (vertical line), as the elbow of the graph.

leveraging LLMs to automatically determine the correct set of parameters to be used to run the container from unstructured documentation pages.

First, for every image we create a "vanilla" container with a default set of `docker run` parameters common across all image executions, thus guaranteeing uniformity across all runs. This strategy, however, will not work for images expecting user-provided parameters in order to execute properly. For example, a WordPress Docker image will not execute successfully unless the user provides the location and credentials of a MySQL server where WordPress' data will be stored. This can result in partial network traces, underestimating the true size of the remote-dependency problem in containerized software.

To tackle this challenge, our second method makes use of the recent advances in Large Language Models (LLMs) and their general availability to the public. In these LLM-assisted runs, we crawl the Docker Hub page of a given image in our dataset and extract the text contained in the "Overview" section, typically describing what functionality the image offers, or what parameters are needed for execution. We use the documentation text as part of an LLM prompt, along with additional text that instructs the LLMs on the task to accomplish and the desired output format. Our goal is for the LLM to interpret a documentation page and produce the appropriate `docker run` command that correctly executes the corresponding Docker image. Table I presents some examples of images and their corresponding LLM-generated commands. Specifically, these are examples of images that appeared to be "inactive" during our vanilla run (i.e. no network activity, indicating an unsuccessful launch) but successfully ran with the LLM-generated command (i.e., we successfully recorded network traffic generated by the container).

In the following, we provide additional details on each of the two container-launching strategies.

Vanilla Run: The vanilla run, in essence, consists of the creation of a Docker network in which an image will reside, selecting a Docker image from our dataset, pulling it from Docker Hub, and launching it with a number of default `docker run` options. Each Docker image is automatically placed by DOCKERGYM in its own Docker network and assigned a unique IP address (e.g., in the `10.0.0.0/8` range), so that we can collect the network traffic associated with that specific image in isolation from all other images that are running in parallel on the same machine.

When running the various Docker images (via the `docker run` command), we chose specific configuration options that were applied across all images to simplify the later analysis of the collected network data. For instance, by using Docker's `--dns-search` option, we rename the search domain to '`local`, to easily distinguish a container's DNS queries to local resources from queries to public domain names. Other decisions involved specifying a public DNS server to be used for domain resolutions and re-using isolated Docker networks between execution batches to improve the overall performance of DOCKERGYM.

We also perform a re-run of the images in our dataset, in which we provide a minimally invasive default entrypoint (e.g. the `/bin/sh` shell) that we know is not going to produce network traffic by itself. This is necessary, as a number of the images in our dataset have no default entrypoint to execute, but have services running in the background that create network traffic. By providing an entrypoint which by itself has no network footprint, we are able to capture this background network traffic that would have otherwise been missed. These images which were only stimulated in the entrypoint re-run are included with the "Vanilla Run" in the respective tables later on.

LLM-assisted Run: The LLM-assisted run follows a setup similar to the vanilla run configuration. However, instead of running an image only using a generic set of parameters that are common across all images, as described above, we add a set of `docker run` parameters that are image-specific and that are automatically derived from the image documentation using LLMs. To this end, we evaluated multiple commercial options through the following heurestic method. We selected 50 popular images, which we could verify had complete documentation and 25 random images from our dataset with more sparse or otherwise limited documentation. We then devised a set of prompts that we tested, and picked the one that was consistently yielding the best results. Next, we went through the options we had selected and sent each one our prompt and

TABLE I: *Examples of Docker images and LLM-generated* `docker run` *commands that enabled a successful execution of the related containers.*

Image Name	LLM-generated Command
roseatoni/foptimum	`docker run [...] -p 9191:9191 -e SPEEDTEST_INTERVAL=1800 -e PING_INTERVAL=15 -e SERVER_LIST=1.1.1.1,8.8.8.8 -d roseatoni/foptimum`
henkallsn/docker-srbminer-multi	`docker run [...] -e ALGO=verushash -e POOL_ADDRESS=stratum+tcp://[XYZ:1234] -e WALLET_USER=[XYZ] -e PASSWORD=[XYZ] -v /tmp/test:/path/to/mount -d henkallsn/docker-srbminer-multi`
asoluter/postgresql	`docker run -e PG_PASSWORD=[XYZ] -e DB_USER=[XYZ] -e DB_PASS=[XYZ] -e DB_ADDR=[XYZ] -e REPLICATION_USER=[XYZ] -e REPLICATION_PASS=[XYZ] -e DB_NAME=[XYZ] -v /tmp/test:/var/lib/postgresql -p 23295:5432 -d asoluter/postgresql`

the respective text from each image's documentation, before finally evaluating their responses. The one that performed the best was OpenAI's GPT-4o model. However we noticed that their GPT-4o-mini model was performing almost as well and had an overall significantly lower cost [2] compared to all the models we tested, so we decided to use it. We did not attempt to train our own model on Docker images or DockerHub in general.

We constrained the model to only add options on top of the pre-existing vanilla run options. The LLM-generated command line options include various environmental variables, port mappings, and volume mounting configurations. We also considered the possibility of a Docker image requiring external resources, such as a database server, for its operations. To be able to meet these external requirements, we deployed a number of commonly used databases and services on a remote host, along with pre-defined user accounts, passwords, and support for remote access. We then instructed the LLM to include that remote host and credentials as environmental variables, when applicable, with the aim that the Docker image would be able to reach out and successfully interact with our services, instead of exiting early due to a failure to reach required external services.

A different challenge we faced with LLM-assisted runs was that some network ports are more commonly used than others, and some images would try to bind to a host port that another image running on the same machine is already using. To avoid port collisions and the resulting unsuccessful image runs, we pre-processed all `docker run` parameters output by the LLM before launching an image, and replaced the host port with a random port number in a range belonging to user-defined ports (10000-65535). The effects of this pre-processing are visible in the third row of Table I, in which the image is binding its port 5432 to host port 23295.

[2]Cost was an important consideration because the input tokens required for this experiment come from the Docker image descriptions from Docker Hub, which can add up to thousands of tokens for a single image.

C. Stimulating the Docker containers

After determining the two main approaches for running containers from the images in our dataset, we considered the possibility of externally "stimulating" the containers and monitoring how they react to these stimuli. The reasoning here is to simulate the real-world behavior of users and services that might interact with the Docker containers. An additional benefit of this stimulation is that it also allows us to gain insights on what services the various Docker images actually run. We detail our two stimulation strategies below.

Network scanning and service discovery: With this option, we aim to scan all 65,536 TCP ports of a given Docker image, fingerprinting any services discovered to be listening on those ports. By covering the entire port range we can also scan for regular services that may be listening on non-standard ports.

We used the `nmap` network scanner in its aggressive mode with the service-fingerprinting option turned on. The option uses built-in modules for popular protocols that attempt to connect to each discovered port, using the appropriate protocol in order to be able to identify the software and version running on it. Our rationale was that Docker images may conduct additional network activity when clients connect to them (e.g. reaching out to a remote log server to add an entry regarding an authentication attempt) thereby allowing us to capture a more complete picture of their true remote dependencies.

Web Server stimulation: The final option focuses more specifically on those Docker images running a web server, information which we obtained through the port scanning and service discovery described previously. This will more closely emulate real users who simply visit the web page served by the Docker image through their browsers. More importantly, it helps activate JavaScript code that would otherwise remain dormant, if accessed by a tool such as "curl" or browsers without JS support. That is, JavaScript code that could be reaching to remote servers for additional code and data and could be potentially exploited to hijack a user's session with that Docker container and possibly escalate to other services inside and outside that container. To minimize noise in our network captures, we used the Mozilla Firefox browser, which allows us to disable captive-portal checks [9], [10].

D. Network Traffic Capture Environment

To facilitate the analysis of network traffic generated by each container, we used programmatically created Docker networks and network bridges, which we can then monitor for incoming and outgoing traffic. Our monitoring was done with standard packet-capturing tools (i.e. `tcpdump`) and produced packet captures for later offline analysis.

To determine for how long each container would need to be monitored to capture its network-level behavior, we randomly sampled a preliminary set of 50 Docker images, executed them, and monitored their behavior over a prolonged period of time. As Figure 5 shows, there is little *new* network activity that occurs after the first 2 minutes of a Docker container's lifetime, at least in our experimental setting in which users do not interact with these services beyond the stimulation described in Section III-C. As a result, to improve DOCKERGYM's efficiency, we stop monitoring a container's network data after 2 minutes, allowing us to better scale our infrastructure and storage space required for the experiments. This is important because, if left unchecked, some Docker images (for instance `hujinbo23/dst-admin-go`) can produce gigabytes worth of repetitive network activity in just a few minutes after launch.

As mentioned earlier, each image we run is placed on its own isolated Docker network so that we can accurately differentiate between traffic originating from simultaneously-executed containers. In this way, we collect one network trace for each distinct Docker image run, which is then analyzed for remote dependencies.

IV. EXPERIMENTAL SETUP

In this section, we first describe our experimental setup and then motivate certain design decisions and configuration parameters that aided our analysis and shortened the time required to conduct our experiments.

A. Scale Considerations

As an initial consideration, we had to pull hundreds of thousands of Docker images through Docker Hub. Due to platform restrictions [6], we had the options of either conducting a few hundred pulls per day through their free-tier API or purchase a Docker subscription. By purchasing and using two subscriptions simultaneously, we were able to achieve a sustainable pull rate of 10K images per day, allowing us to rapidly prototype our system and test it at scale, as well as conduct multiple followup experiment runs with different launching plugins and stimulation strategies.

In addition, our infrastructure also needs to be able to execute a commensurate number of images each day, stimulate them, and collect their network traffic. For that reason, we opted to scale horizontally, both in terms of machines used, as well as how many times each machine is running the pipeline in parallel. As the pipeline is only bound by the input of Docker images, by giving each instance a different subset of images to execute, we can scale it up as necessary without encountering any overlap in terms of executed images.

B. Experiment Runs

We performed five types of runs in total. Two of those used the "Vanilla" launching plug-in and either "Vanilla" or Network Scanning for the stimulation component. The other three experiment runs had LLM as its launching plug-in, and each different run used a different stimulation option ("Vanilla", Network Scanning, and Browser Automation).

The first two runs are conducted over our entire dataset of 200K images, as there were no restrictions on them. For the LLM-Assisted runs on the other hand, we can only use Docker images that include some documentation on their Docker Hub page. This turns out to be about 45K images of the original 200K. Furthermore, the final run (LLM-assisted + Selenium stimulation) is performed only for the subset of images that had both documentation (as required by the LLM part) and also run a web server. The knowledge of which image hosted a web server is obtained by examining the output of the Network Scanning-stimulated runs executed previously.

C. Computing Experiment Results

Our goal when setting up the experiments was to focus on behavior that clearly indicated misconfigurations (i.e. behavior that *could* lead to exploitation) or outright vulnerabilities (i.e. behavior that *does* lead to exploitation). To analyze the PCAP files collected by DOCKERGYM, we built a set of custom scripts that perform queries on the captured network packets. Our queries extract all the relevant information from TCP, UDP, and ICMP packets with a particular focus on the DNS, HTTP, and TLS protocols. With that in mind, we set up the following analyses:

DNS-related traffic: By definition, when an image makes a public DNS query as part of its normal operations, that makes it dependent on external services. From a strict threat-modeling perspective, the mere existence of this traffic may constitute unwanted behavior. That aside, we follow up on the DNS queries the images make by investigated the actual resolved domain names. We specifically look for domains that yield DNS errors, such as NXDOMAIN (indicating a possible domain expiration) and SERVFAIL (indicating a misconfigured nameserver that cannot answer the image's resolution requests). We also extract the effective second-level domain label for each queried domain name and explore whether those domains are parked or are listed as for sale. In both cases, motivated attackers can purchase the domains from their current owners and may be able to use them to takeover sensitive containerized applications.

Hijackable domains: Since some of the aforementioned domains were in fact registrable, we registered them for ourselves, pointed them to a server we control, and monitored the incoming traffic, aiming to understand more about how in-the-wild Docker containers that interact with them. Specifically, given a registered domain d, we refer to its corresponding server as W_d, whose IP address resolves from d. Given an in-the-wild client C that contacts W_d, we collect the traffic sent by C to our server.

A challenge we face here is represented by the "noise" stemming from generic crawlers that roam the web [20], [25] or even certificate transparency bots [22], [39] that visit servers as soon as a related TLS certificate is issued and logged. It can therefore be difficult to discern whether an incoming request is due to a running Docker container that is reaching out to our infrastructure vs. a generic web/port-scanning bot.

To resolve this, we devised a two-layer approach. First, we record the software description declared as part of the User-Agent field in the traffic generated as we ran the Docker images within our measurement pipeline. If the User-Agent advertised by an in-the-wild client does not match the User-Agent previously recorded in the lab, we can exclude the client from further consideration. Second, for a registered domain d, we set up a separate "sibling" web server $W_d^{'}$, which is hosted in the same address range as our original server (the same /24 subnet) but whose IP address does not resolve from d (or any other domain name we registered). Additionally, $W_d^{'}$ is set up with the same open ports as W_d. Following the setup of So et al. [42] and their placebo hosts, this second server's purpose is to act as a "baseline," whereby if we observe the same clients contacting both servers, we can conclude that they are scanners and disregard them.

It is important to note here that, while the possibility of directly responding to incoming requests to our server could lead to a much more streamlined bot filtering process, we rejected it on ethical grounds. It would be impossible for us to predict how a given system might react to our network responses and therefore we could not know if our actions would cause damage to a remote system or organization. We therefore allow the clients to connect to our server and, for TCP-based protocols, we tear down the connection by sending a RST packet after a request is received. We elaborate upon our ethical considerations in Appendix A.

HTTP Traffic: HTTP is not a secure protocol, with its plaintext traffic making it vulnerable to man-in-the-middle and eavesdropping attacks [15]. The obvious insecurity of the protocol aside, we analyze the captured traffic from the Docker images for any instances of it and further break down the traffic that yields non-200 codes (3XX, 4XX, and 5XX). Of special interest are the 3XX redirects, and more specifically how many requests, if any, are redirected to HTTPS.

TLS Traffic: Even though TLS connections from Docker containers to the outside world are better than plaintext protocol, they may still be connecting with outdated TLS servers supporting vulnerable TLS versions and cipher suites. We therefore collect the version of the TLS servers they connect to and analyze their offered certificates.

Miscellaneous Data: Finally, we keep track of statistics related to the Docker images themselves, such as how many times they were "pulled" or when they were last updated. While the pull count is a proxy of an image's popularity (the more times it is pulled, the more containers based off of that image are likely to be executing in the wild), we are careful not to overly rely on it in terms of vulnerability impact. We reason that images with low number of pull counts can still be deployed in critical environments where a single exploited container can be the beginning of an attack that will affect millions of users and their data. For example, the 2017 Equifax hack that affected hundreds of millions of US citizens started with a single vulnerable server that was running an out-of-date version of Apache Struts [12].

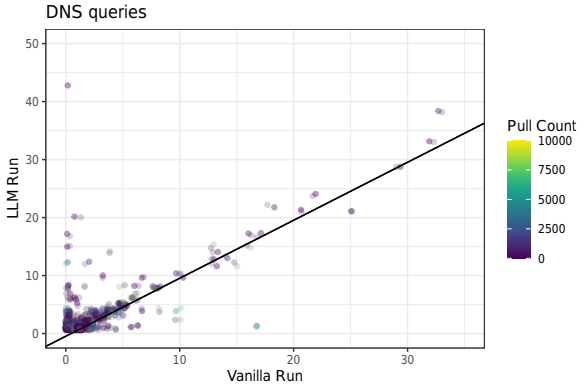

Fig. 6: *Comparison of the DNS queries made per image on the vanilla run and the LLM-assisted run.*

V. RESULTS

In this section, we analyze the results of our distinct runs, comparing and contrasting between them, aiming to understand the remote uncontained dependencies of popular Docker images.

A. Image Runs and Internet Traffic

Our first goal is to evaluate how many Docker images connect to the Internet in the first place. We execute our five distinct runs as described in Section IV. These runs are performed sequentially, with a period of one to two weeks between them. The first two "vanilla" runs, which also involved the most images, lasted about three weeks each. The next two runs were the service-agnostic LLM-assisted runs, which took about a week each, and the final LLM-assisted run focusing on images deploying web servers lasted around one day. In total, given our infrastructure and Docker API quotas, our runs required approximately four months of data gathering and execution time.

We also note that not all images in our dataset were executed during these runs, because some of them were built for CPU architectures different from our system's architecture (e.g. they required ARM CPUs), some were taken down in-between runs, or failed to execute because of instrumentation issues on our end (these issues were minor and can be addressed with additional engineering efforts). For our results to be consistent and comparable across runs, we focus on the intersection of the images that were successfully executed across our different runs, and will only report measurements on those. For that reason, for the first two vanilla runs we focus on 173,683 images out of the original 200K. Similarly, for the two service-agnostic LLM-assisted runs we focus on 41,968 out of the

TABLE II: *Traffic recorded across our image runs. The presented statistics are based on the number of distinct images that either generally reach out to the public Internet (TCP/UDP traffic) or specifically attempt to resolve one or more domain names.*

	Image Total	Images w/ Internet Traffic	Images w/ Public DNS	SERVFAIL	SOA + NX-DOMAIN	SOA + NOERROR	Re-registrable
Vanilla Run	173,683	23,189 (13.4%)	10,188 (5.9%)	52 (0.5%)	985 (9.7%)	309 (3.0%)	82 (0.8%)
Vanilla Run + Net. Scan.	173,683	22,353 (12.9%)	9,823 (5.7%)	60 (0.6%)	950 (9.7%)	299 (3.0%)	86 (0.9%)
LLM-Assisted	41,968	4,086 (9.7%)	1,948 (4.6%)	19 (1.0%)	220 (11.3%)	78 (4.0%)	47 (2.4%)
LLM-Assisted + Net. Scan.	41,968	4,072 (9.7%)	2,085 (5.0%)	16 (0.8%)	170 (8.2%)	71 (3.4%)	40 (1.9%)
LLM-Assisted + Web Crawler	723	74 (10.2%)	49 (6.8%)	0 (0%)	0 (0%)	1 (2.0%)	0 (0%)

original set of about 45K (i.e. the Docker images that included some level of documentation that could be provided as input to an LLM). We discuss the reasons for these discrepancies more in-depth in Section VII.

As shown on Table II, we find that a large number of Docker images reach out to the public Internet in the form of TCP or UDP requests. Over 20,000 images, or about 13%, of our Vanilla runs have some type of Internet traffic. While that number decreases to about 10% in our LLM-assisted runs, this can be largely attributed to the LLM output occasionally "tripping up" the execution of the images, not allowing some of them to properly execute. As we describe later in this section, when the LLM model works as intended, we observe a definitive increase of network activity. Comparing the overall numbers between the Vanilla/LLM-Assisted runs and then the re-runs including the Network Scanning module, we can see that there is a slight decrease in the number of images that attempt to access the internet. Possible explanations include non-deterministic image executions or a strain to our computing resources, leading to the mis-execution of some images that would otherwise have reached out to the Internet. We also note that these numbers explicitly exclude our own activity – such as when using a network scanner to stimulate the executing containers.

B. DNS Traffic

By looking at Table II, it is immediately apparent that a sizable number of Docker images have some type of remote dependency, as approximately 5% of the images in our dataset during the vanilla run make some type of DNS query to the outside world.

In Figure 6, we can see the images that exist in both the Vanilla and LLM run, and made a DNS query in either one (or both). The figure gives us an insight in how LLM assistance can help more properly execute Docker images. For instance, of the 1,948 (4.6%) images that correspond to images with DNS queries, only 1,376 had any DNS queries in the original vanilla run. This means that, when the LLM-Assisted version of our pipeline is successful at finding the correct parameters necessary to launch specific Docker images, it enables 50% more Docker images to make DNS queries. Any images that are in the x=0 coordinate in Figure 6 present cases in which the LLM-Assisted Run enabled the images to perform DNS queries when the Vanilla run was unable to. Examples of some of these can be viewed on Table I, where we can see that for the `henkallsn/docker-srbminer-multi` image, the LLM provided the parameters for successful cryptocurrency mining, or on `asoluter/postgresql`, the parameters necessary to access our remote Postgres Database. On the flip-side, there were also 1,713 cases where the original Vanilla run had DNS queries, but the LLM-assisted run failed to produce any network traffic. Upon manual inspection of the LLM output for some of those images, this occurred due to the images being improperly launched by the LLM. When considered together, this means that both unassisted and LLM-assisted runs should be utilized, when one wants to maximize the number of Docker images that can be successfully launched and evaluated. We were also able to confirm that there were images in the LLM-assisted Web Crawler run that were active during only that run, hinting that certain images contain services that would reach out to the Internet when interacted with properly.

TABLE III: *The most popular effective second-level domains (TLD+1s) queried for by images in our "vanilla" runs, and their percentage value when compared to the total images making DNS queries.*

DNS query	Images querying domain
npmjs.org	1,975 (19.4%)
jenkins.io	1,053 (10.2%)
jenkins-ci.org	768 (7.5%)
github.com	637 (6.3%)
hashicorp.com	391 (3.8%)
influxdata.com	363 (3.6%)
githubusercontent.com	363 (3.6%)
amazonaws.com	335 (3.3%)
mongodb.net	334 (3.3%)
gravatar.com	216 (2.1%)

By attempting to resolve all these domains and analyzing the outputs of our resolution request, we discovered that approximately 10% of the DNS-utilizing Docker images make

TABLE IV: *Analysis of HTTP traffic across runs, showing the number of images in each run. The presented statistics highlight the number of distinct Docker images that performed HTTP requests and the non-standard status codes associated with these requests.*

	Image Total	Images w/ HTTP requests	3XX	Redirect to HTTPS	4XX	5XX
Vanilla Run	173,683	1,244 (0.7%)	756 (60.8%)	159 (21.0%)	175 (14.1%)	13 (1.0%)
Vanilla Run + Net. Scan.	173,683	1,340 (0.8%)	825 (61.6%)	176 (21.3%)	176 (13.1%)	14 (1.0%)
LLM-Assisted	41,968	329 (0.8%)	150 (45.6%)	74 (49.3%)	58 (17.6%)	3 (0.9%)
LLM-Assisted + Net. Scan.	41,968	303 (0.7%)	142 (46.9%)	70 (49.3%)	54 (17.8%)	3 (1.0%)
LLM-Assisted + Web Crawler	723	8 (0.8%)	4 (50%)	1 (25.0%)	3 (37.5%)	0 (0%)

at least one request to domains yielding NXDOMAIN errors. We discuss the implications of this finding in the next section.

As discussed previously, even when the images reach out to domains that work as intended, this can have adverse effects regarding the image's deterministic outcomes. Even though requests to, for instance, `ubuntu.com` can be considered "standard", by looking at Table III we discover a vastly different set of popular remote dependencies. Most images that make DNS queries reach out to dynamic content from websites like `npmjs.org` or `github.com` that have recently been linked to supply chain vulnerabilities [21], [33], [46].

To understand the popularity of the domain names resolved by the evaluated Docker containers, we search for these domains in the Tranco top 1 million most popular websites list [35]. Of all the domains, 70% are present in the Tranco list. Figure 7 shows their distribution, where we observe that more than 50% of the queried domains have a Tranco rank greater than 130K, and therefore correspond to lesser known domain names. Importantly, 30% of the resolved domain names are not included in that list thereby signifying that these are significantly unpopular domain names. As prior research has established [17], [18], [34], [44], all other things being equal, less popular sites with fewer resources are more likely to be vulnerable to attacks and thereby abusable as a stepping stone in supply-chain attacks to reach critical Docker containers.

1) Hijackable Domains: One subcategory of DNS traffic that warrants particular attention is that of attempting to resolve domain names that are actually available for registration. This group of domains is straightforwardly exploitable, since attackers can just re-register them and point them to infrastructure under their control. To pinpoint these domains in our dataset, we started by using the same list containing all the domains our Docker images queried for via DNS. We then extracted the TLD+1 of those domains – so, in the case of `sub.example.com`, we would extract `example.com` – and performed `dig` queries on them, looking for responses containing both a "Non-Existent Domain" (NXDOMAIN) and "Start of Authority" (SOA) field, which in this case would contain the top level domain. This means that the domain is either unconfigured or expired. Looking deeper into what types of domains these are, we find two general categories present in our dataset.

The first category contains domains that were set to a poorly-chosen default value by the Docker image publisher,

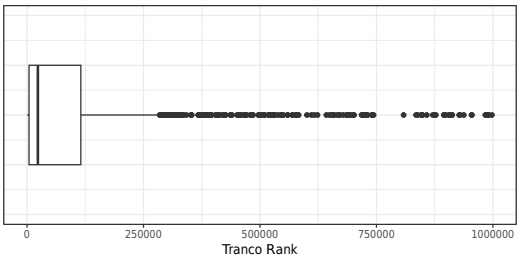

Fig. 7: *Tranco ranks of the websites queried for via DNS in our vanilla run. 30% of the resolved domain names are not included in the top 1 Million Tranco domains.*

meant to be changed by the end user once they eventually set their system up. Examples of this include domains such as `sql.data` or `backup.network`. In this case, both `.data` and `.network` are legitimate generic top level domains (gTLDs) [7]. The second category is composed of domains that simply expired over time.

Looking at Table II, we can see that for the vanilla-launched runs, about 1% of images queried domains that were re-registrable, or "hijackable". When looking at the LLM-assisted runs, the relative number increases to about 2%. We then selected a number of domains and registered them ourselves, to better understand their interactions with the Docker images.

By analyzing the Docker image network logs of the relevant images, we can infer what the Docker images are looking for in those domains. One Docker image, for instance, is reaching out to port 5672 of a specific domain. We in turn open port 5672 on the server that the domain – which we are now in control of – resides in, and log all incoming connections to that port.

We set up our data collection for 4 domains, which are contacted by 3 distinct Docker images. We collected data for a period of 2 months, and apply our two-layer filtering as outlined in Section IV. We find that there were 311 unique hosts which passed our filters in that period of time, indicating the danger that could arise were these domains compromised by an adversary.

C. HTTP Traffic

The use of unencrypted protocols is a security liability in and of itself. In analyzing Docker image traffic, we observe thousands of images making HTTP requests over the public Internet (Table IV).

TABLE V: *The most common filetypes requested via HTTP by Docker images during our "vanilla" runs, and their percentage value compared with the total images making HTTP requests.*

Filetype	Images requesting filetype
JSON	583 (47.9%)
gz	90 (7.2%)
HTML/XML	80 (6.4%)
zip	53 (4.3%)
txt	53 (4.3%)

By focusing first on the first four runs, we see that most images exhibiting HTTP traffic are getting 3XX-type responses. Through sampling of these redirects, we find that they tend to redirect to the same TLD+1, with a small number of redirects leading to specific IP addresses (for example, `www.payara.fish/phonehome` redirecting to `http://52.213.212.227:8083/phonehome` – this is an IP address different to the one of the server responding with the redirect). Surprisingly, most of the servers performing redirects do so to other HTTP resources, instead of the more secure, encrypted HTTPS protocol. The Docker images using redirections to HTTPS endpoints give the appearance of security but are hijackable through traditional SSL stripping attacks [29].

4XX-type responses indicate a client-side error with the most popular code (HTTP 404) indicating that a specific resource cannot be found at the requested URL. These errors are clear signs of misconfiguration, with the extreme being that the requested domains are now under the control of a different party who does not host the same resources as the previous owner. Particularly, in the case of parked domains, motivated attackers can purchase the domains from their current owners and thereby abuse their residual trust to potentially compromise vulnerable Docker images. In our dataset, we find 63 parked domains, which are in addition to the number of domains that we found to be freely available for registration.

As part of our analysis, we also collected a list of the file types most commonly requested via HTTP by the Docker images. This list can be seen in Table V, dominated by "JSON" filetype requests - half the images in our dataset that make HTTP requests, make one for a JSON file. We further explored the requests aimed at `zip` and `gz` filetypes and found that the vast majority of these compressed files contain executables. These compressed binaries fetched by Docker images over the network pose a clear threat, particularly since they are being downloaded over HTTP and can therefore be altered in transit by MITM attackers.

D. TLS Traffic and Certificate Validity

When analyzing TLS, we are limited to the unencrypted traffic we can access, in the context of TLS handshakes. We focus on the TLS version accepted by the remote server, as well as the validity of the presented certificate chain.

In our experiments across all five distinct runs, we note the same pattern when it comes to TLS traffic: A majority of Docker images communicate over the TLSv1.2 protocol, with a sizable minority (≈25%) using TLSv1.3. There are a few outliers in which we noted the usage of TLSv1 and even SSLv2/v3, however upon manual inspection those images were flagged as such due to malformed packets. In other words, when a Docker container does employ TLS, we find that the remote TLS servers are mostly up to date.

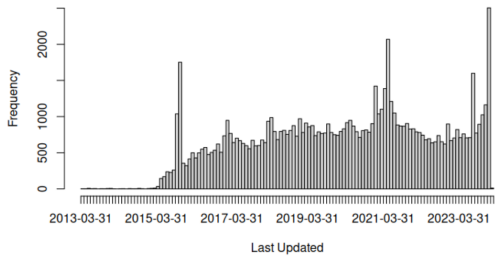

Fig. 8: *Histogram of the dates an image in our dataset has been last updated by the owner of its repository.*

Regarding the validity of the offered certificates, we find that about 3% of images across all runs have at least one expired certificate present in their certificate chain. By manually sampling the 90th percentile of those images, we confirm that few of these certificates are leaf certificates, with the majority being intermediate.

E. Outdated Software

As shown in Figure 8, many Docker images in Docker Hub have not been updated for years, which attackers can compromise through their outdated software. As part of our analysis, we explored the software versions declared via the User-Agent string of the HTTP Requests these images made, to understand the extent of this phenomenon. These include software like wget, curl, python requests, but also more complicated pieces of software, such as Sonarr.

We found that in our initial vanilla run, out of 22,842 images that reach out to the Internet, 1,034 (or 4.5%) are doing so by using outdated software. In our LLM-assisted run, out of 4,086 images that connect to the Internet, 217 (5.3%) do so over outdated software.

F. HTTP(S) Sampling

In an effort to better understand how these images interact with the respective HTTP/HTTPS resources, we manually sampled Docker images that sent HTTP(S) traffic. When it came to the images making HTTP requests, we looked at the plaintext communication between our sampled Docker image and the server to identify the purpose of the communication. For TLS-encrypted traffic, we had to engage in TLS termination in order to obtain decrypted requests and responses. We used `mitmproxy` to reroute and decrypt the traffic through a trusted certificate that we copy over to the Docker image. We focused this sampling strictly on the top 1% most popular images in our dataset.

We find that both the HTTP and HTTPS traffic in Docker images largely falls into the same general categories, which can be described as follows:

- **File downloads:** As shown in Table V, the images in our dataset that make HTTP requests, sometimes attempt to download some type of compressed file. For example, an image in our dataset reaches out to an IoT manufacturer's website over HTTP, requesting a software update file, which it then downloads in `.gz` format.
- **Jenkins images:** Jenkins software images are the ones most commonly making HTTP requests, in which they request `JSON`-type files. On Table V we can see that half of the images that make HTTP requests overall, attempt to request a JSON file. Through a closer look, 540 of those images are Jenkins-adjacent. In the context of Continuous Integration / Continuous Deployment (CI/CD) applications, it is worth noting that by controlling or manipulating the JSON file that commonly provides configuration of the pipeline, the attackers can control the CI/CD pipeline itself.
- **Linux distribution updates:** This category amounts to images running the equivalent of a `apt-get update/upgrade`. The image updates the list of software (or the software itself) before continuing with its operations. Such images may make requests to online resources not only located on well-known domains such as `ubuntu.com`, but less popular ones as well which, if compromised, could be abused for supply-chain attacks.
- **Online services:** The images in this category reach out to various API endpoints. For example, an image in our sample made requests to `ridetimes.co.uk` receiving event-scheduling information. Some of these endpoints may be less well-managed or may implement much weaker security controls than more reputable endpoints.

Most of these categories and images that perform some type of HTTP or HTTPS requests download software, software libaries, or configure their software based on remote resources. As such, they expose the Docker images to potential abuse from a malicious adversary in a supply-chain context, as discussed in our threat model (Section II-B).

G. Impact

An additional dimension that one could add to these metrics is that of the "pull count". As mentioned in Section IV,we posit that one should not be overly reliant on the pull counts as they pertain to vulnerability impact. Regardless, all else being equal, a more popular image getting compromised will lead to more users getting compromised. For that purpose, we evaluate the pull counts of the five most popular images that are affected by our most critical findings (e.g. HTTP traffic, NXDOMAIN errors) and find that they are in the millions and tens of millions. We further evaluate the median pull count for all the images affected and find it to be closely tracking the overall median of all images of around 100, visible in Figure 4. A more detailed breakdown of these findings can be found in Appendix C.

VI. LIMITATIONS

As with any real-life system, the remote-dependency analysis pipeline described in this paper has some limitations, which can be addressed in future work:

Incomplete stimulation: As the first large-scale study of this type, we made the decision to take a service-agnostic approach to stimulating the Docker images in our dataset. We treated each Docker image as a complete black box and performed full service discovery on every image.

Furthermore, our LLM-assisted runs were successful in that we were able to run Docker containers with the proper command-line parameters and correct environment variables required to provide the containers with the necessary external "helper" services. Even though this approach yielded a significant improvement over our baseline runs (e.g., increasing the number of images that make queries to the internet by 50% in some cases), it is far from perfect. For example, Docker images *without* documentation were not part of our LLM-Assisted runs as we did not have any context to provide to the LLM. Similarly, although we provided a number of helper services (e.g. web servers, database servers, etc.) that the LLM could plug into the appropriate command-line parameters and environment variables, our set of servers is unlikely to fully satisfy the diverse requirements of hundreds of thousands of Docker images.

One possible idea for future work is to therefore employ a gray-box-style analysis (instead of a black box one) where the code and data available in a Docker image (e.g. its Dockerfile) could be analyzed in search for hints for execution and stimulation.

Images that do not execute: As a tie-in to the previous point, some of the images we discovered when curating our dataset did not successfully execute in our analysis environment. There were various reasons that contributed to these missing executions, including a mismatch between the expected and offered CPU architecture. One interesting finding (which was recently corroborated by an industry report [36]) was the abuse of Docker Hub for Search Engine Optimization purposes. In these cases, attackers create dummy images whose sole purpose is to link to other third-party sites and therefore do not carry any code that we can download and execute.

Besides the aforementioned reasons, some of the missing executions was due to images being taken offline (perhaps because they were malicious) between runs and to some operational instability of a research framework that aims to automatically execute and analyze hundreds of thousands of images from a multitude of different developers. In this study, we accounted for these transient issues by focusing on the images that were successfully executed across all runs. The framework and execution pipeline could be further improved with additional engineering efforts to further increase image execution coverage.

VII. DISCUSSION AND FUTURE WORK

Dealing with remote dependencies: The biggest danger due to remote dependencies occurs when end users are unaware of the potential of non-deterministic runs. Even if they have no alternative but to use a given Docker image, users who are aware of the remote dependency pitfalls may be able to plan around them.

One of the pragmatic solutions that we envision, is for platforms such as Docker Hub to scan the layers of an image for existing remote dependencies, and notify end users. This scanning can be done either statically or potentially using a dynamic analysis system such as the one presented in this paper. The discovered dependencies can be included on the Docker Hub website, allowing users to make an informed decision on whether they can trust the domain names being relied upon by a specific Docker image.

From an end-user perspective, one should always be wary of executing third-party Docker images in their production environment. A Docker image executed in "privileged" mode, for instance, can modify iptables rules, kernel parameters or even load/unload kernel modules. In effect, the end user could treat an image the same way as one would treat potential malware. It can be statically analyzed by inspecting its layers and metadata, as well as be dynamically executed with limited capabilities, or in an isolated environment such as a VM. One could then examine the services as well as potential network communications it attempts to make. In lieu of that, an end user could attempt to force restrictive network-based access control to it to ensure that the image communicates only with services and over protocols that the user expects it to.

Image tag consistency: In this paper, we showed that even if a user "pulls" an image from Docker Hub and gets the exact same code they had gotten before, this code can still be vulnerable to supply-chain attacks if it makes use of remote dependencies. As we described in Section II, these dependencies can be both explicit or implicit, depending on the component that reaches out to the public Internet and its position in the lifetime of a Docker container.

Beyond these dependencies, however, we observe that the use of tags remains an understudied aspect of the Docker ecosystem, which should be studied in future work. Specifically, we argue that Docker Hub's support of tags is likely different from how users may be understanding them. That is, the owners of a Docker image can push arbitrary updates under a specific tag, even if the tag itself would indicate a fixed software version (e.g. `nginx:1.25.4`). This can create confusion, as developers will commonly integrate the tag, instead of the Docker image's digest, in their code – and thus "pull" any untested image updates along with it, potentially breaking their system. We expect that not all users understand these mechanics, particularly when some tags (e.g. `latest`) are meant to indicate that that tag will always point to the most recent version of an image.

A possible solution for users who really want to ensure that they always pull the same image as before is to instead rely on Docker digests (e.g. `nginx@sha256:f9c014f9ae6...`) instead of tags. While these digests are already supported in Docker Hub, they are much less prominent than tags. It therefore may be worthwhile to explore UI-level nudges to make developers aware of this separate image-addressing mechanism.

Ethical considerations: During our experiments we took a number of precautions to minimize the risk of harm to users and systems. In the vast majority of experiments, we did not interact with remote systems or users since all Docker images were downloaded, executed, and stimulated on our servers and infrastructure.

Our experiment with re-registered domains resolved by Docker images of interest was the only experiment where we interacted with the outside world and where existing Docker containers contacted our servers. Beyond registering these domains and obtaining certificates from them, we never advertised them to the outside world or attempted to attract traffic to them. When we did receive connections, we took the conservative approach of logging the incoming request, then tearing down the connection instantly. For this reason, we are confident that our registration of expired domains associated with Docker images did not negatively impact users or systems. A more detailed breakdown of our precautions regarding re-registered domains can be seen in Appendix A.

We also made a best-effort attempt at reaching out via any method possible (e-mail, social media) to the owners of the 100 most popular Docker repositories we found to be problematic, such as images that download software over HTTP or reach out to expired domains. We have received one response so far, which we include in Appendix B, in which the developer notified us they took steps to secure their application as well as contact DockerHub to issue a takedown for what turned out to be an unauthorized redistribution of their software.

VIII. RELATED WORK

Despite the fact that Docker containers have been around for a decade [30], to the best of our knowledge this paper is the first to perform a large scale analysis of the network activity of Docker images, focusing on remote dependencies. Prior work has focused on container security [13], [26], [38], either through the lens of Docker's internal security [19], [32], or through how Docker containers interact with a given host operating system [43].

Dahlmanns et al. [16] showcase how compromised Docker images can have adverse real-world effects by analyzing over 300,000 images from Docker Hub and 8,000 from private registries, scanning them for private keys and API secrets that users mistakenly share along with their containers. They discover that over 50,000 private keys and over 3,000 API secrets are leaked, breaching confidentiality. They further find certificates issued in the wild relying on compromised keys from their dataset, as well as over 275,000 TLS and SSH hosts using leaked private keys for authentication.

Liu et al. [28] performed a large-scale in-depth security analysis of 10,000 popular Docker images and scan them for vulnerabilities, studying the vulnerability window of a given image before that image is patched. They uncover that vulnerability patching of software in Docker images is delayed by months or even years, on average. They also find a subset of outright malicious images uploaded to Docker Hub.

Shu et al. [40] create a scalable Docker image vulnerability analysis framework which statically analyzes images found on Docker Hub. They study 100K images for multiple tags (resulting to a total of 440K unique image and tag combinations) and find that images are on average not updated for hundreds of days, while vulnerabilities propagate from parent images to child images. While this is, to our knowledge, the largest-scale analysis of Docker images until know, the authors rely strictly on static analysis of those images and did not perform any type of network monitoring.

Campo et al. [14] explore workflows to address container image vulnerabilities with known fixes. In doing so, they first assess a dataset of popular images, scanning them for vulnerabilities. They find that these images have a median of 143 vulnerabilities, most of which have readily-available patches. They find, however, that there are technical barriers to patching off-the-shelf images and that Docker Hub itself misstates the security of these images. They also note that Docker Hub does not list vulnerabilities for older image tags, potentially confusing users about their (in)security.

The issue of remote dependencies has been analyzed in different non-Docker settings including websites, HTTP headers and even malware [23], [34]. Finally, relating to this work is research performed on how adversaries leverage expired or otherwise re-registrable domains [24], [31], [41].

IX. CONCLUSION

In a world of continuous integration and ever more complicated systems, developers have come to rely more and more on automation. It is therefore of utmost importance that each small piece of a large software pipeline works as intended. In this paper we curated a dataset of 200K images publicly available on Docker Hub and designed an analysis pipeline capable tailored to the discovery of remote dependencies. We fed our dataset to our dynamic analysis system, DOCKERGYM, multiple times, each time focusing on a different aspect of launching and stimulating these Docker images, including the use of state-of-the-art Large Language Models to help us run these images with all their expected parameters. Among others, we discover that more than 10% of popular Docker images send traffic to the public Internet and 5% resolve one or more domain names as part of their operations. Many of these domains belong to popular endpoints (e.g. Amazon Web Services and GitHub) which however are highly dynamic and can therefore radically change the code and data of their downstream dependents. We quantify this reliance on external services finding misconfigured servers, plaintext HTTP, and domains that can be readily hijacked by attackers and be exploited to reach all Docker images that depend on them.

AVAILABILITY

To inspire additional research in the area of containerized software and supply-chain attacks we are open-sourcing our dataset, as well as DOCKERGYM at the following Github repository: https://github.com/uncontained-danger/artifacts

ACKNOWLEDGMENTS

We thank the anonymous reviewers and shepherd for their helpful comments. This work was supported by the Office of Naval Research (ONR) under grant N00014-24-1-2193 and the National Science Foundation (NSF) under grants CNS-1941617, CNS-2126641, CNS-2126654 and CNS-2211575.

REFERENCES

[1] Cve-2017-5123. https://nvd.nist.gov/vuln/detail/CVE-2017-5123. Online.
[2] Cve-2019-14271. https://nvd.nist.gov/vuln/detail/CVE-2019-14271. Online.
[3] Cve-2019-5736. https://nvd.nist.gov/vuln/detail/CVE-2019-5736. Online.
[4] Docker adoption. https://www.datadoghq.com/docker-adoption/. Online.
[5] Docker customer success. https://www.docker.com/customer-success/. Online.
[6] Docker download rate limit. https://docs.docker.com/docker-hub/download-rate-limit/. Online.
[7] Icann list of top-level domains. https://data.iana.org/TLD/tlds-alpha-by-domain.txt. Online.
[8] Investigation updates and community call to action. https://x.com/safe/status/1897663514975649938. Online.
[9] Network portal detection on chromium. https://www.chromium.org/chromium-os/chromiumos-design-docs/network-portal-detection/. Online.
[10] Network portal detection on firefox. https://support.mozilla.org/en-US/kb/captive-portal. Online.
[11] Safewallet confirms north korean tradertraitor hackers stole $1.5 billion in bybit heist. https://thehackernews.com/2025/03/safewallet-confirms-north-korean.html. Online.
[12] Apache Foundation. Apache Struts Statement on Equifax Security Breach. https://news.apache.org/foundation/entry/apache-struts-statement-on-equifax, 2017.
[13] Thanh Bui. Analysis of docker security. *arXiv preprint arXiv:1501.02967*, 2015.
[14] Frank Campo, Howe Wang, and Joel Coffman. Exploring solutions for container image security. In *2023 IEEE 14th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, pages 82–88. IEEE, 2023.
[15] Mauro Conti, Nicola Dragoni, and Viktor Lesyk. A survey of man in the middle attacks. *IEEE communications surveys & tutorials*, 18(3):2027–2051, 2016.
[16] Markus Dahlmanns, Constantin Sander, Robin Decker, and Klaus 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*, pages 797–811, 2023.
[17] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. The cookie hunter: Automated black-box auditing for web authentication and authorization flaws. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1953–1970, 2020.
[18] Tom Van Goethem, Ping Chen, Nick Nikiforakis, Lieven Desmet, and Wouter Joosen. Large-scale Security Analysis of the Web: Challenges and Findings. In *Proceedings of the 7th International Conference on Trust & Trustworthy Computing (TRUST)*, pages 110–126, 2014.
[19] Zhiqiang Jian and Long Chen. A defense method against docker escape attack. In *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy*, pages 142–146, 2017.
[20] Md Abu Kausar, VS Dhaka, and Sanjeev Kumar Singh. Web crawler: a review. *International Journal of Computer Applications*, 63(2):31–36, 2013.

[21] Igibek Koishybayev, Aleksandr Nahapetyan, Raima Zachariah, Siddharth Muralee, Bradley Reaves, Alexandros Kapravelos, and Aravind Machiry. Characterizing the security of github {CI} workflows. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2747–2763, 2022.

[22] Brian Kondracki, Johnny So, and Nick Nikiforakis. Uninvited Guests: Analyzing the Identity and Behavior of Certificate Transparency Bots. In *Proceedings of USENIX Security Symposium (USENIX Security)*, 2022.

[23] Deepak Kumar, Zane Ma, Zakir Durumeric, Ariana Mirian, Joshua Mason, J Alex Halderman, and Michael Bailey. Security challenges in an increasingly tangled web. In *Proceedings of the 26th International Conference on World Wide Web*, pages 677–684, 2017.

[24] Chaz Lever, Robert Walls, Yacin Nadji, David Dagon, Patrick McDaniel, and Manos Antonakakis. Domain-z: 28 registrations later measuring the exploitation of residual trust in domains. In *2016 IEEE symposium on security and privacy (SP)*, pages 691–706. IEEE, 2016.

[25] Xigao Li, Babak Amin Azad, Amir Rahmati, and Nick Nikiforakis. Good bot, bad bot: Characterizing automated browsing activity. In *2021 IEEE symposium on security and privacy (sp)*, pages 1589–1605. IEEE, 2021.

[26] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A measurement study on linux container security: Attacks and countermeasures. In *Proceedings of the 34th annual computer security applications conference*, pages 418–429, 2018.

[27] Guannan Liu, Xing Gao, Haining Wang, and Kun Sun. Exploring the unchartered space of container registry typosquatting. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 35–51, 2022.

[28] Peiyu Liu, Shouling Ji, Lirong Fu, Kangjie Lu, Xuhong Zhang, Wei-Han Lee, Tao Lu, Wenzhi Chen, and Raheem Beyah. Understanding the security risks of docker hub. In *Computer Security–ESORICS 2020: 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14–18, 2020, Proceedings, Part I 25*, pages 257–276. Springer, 2020.

[29] Moxie Marlinspike. More tricks for defeating ssl in practice. *Black Hat USA*, 516, 2009.

[30] Dirk Merkel et al. Docker: lightweight linux containers for consistent development and deployment. *Linux j*, 239(2):2, 2014.

[31] Tyler Moore and Richard Clayton. The ghosts of banking past: Empirical analysis of closed bank websites. In *International Conference on Financial Cryptography and Data Security*, pages 33–48. Springer, 2014.

[32] Amith Raj MP, Ashok Kumar, Sahithya J Pai, and Ashika Gopal. Enhancing security of docker using linux hardening techniques. In *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, pages 94–99. IEEE, 2016.

[33] Siddharth Muralee, Igibek Koishybayev, Aleksandr Nahapetyan, Greg Tystahl, Brad Reaves, Antonio Bianchi, William Enck, Alexandros Kapravelos, and Aravind Machiry. {ARGUS}: A framework for staged static taint analysis of {GitHub} workflows and actions. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6983–7000, 2023.

[34] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 736–747, 2012.

[35] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. *arXiv preprint arXiv:1806.01156*, 2018.

[36] Andrey Polkovnichenko, Brian Moussalli, and Shachar Menashe. JFrog Security research discovers coordinated attacks on Docker Hub that planted millions of malicious repositories. https://jfrog.com/blog/attacks-on-docker-with-millions-of-malicious-repositories-spread-malware-and-phishing-scams/, 2024.

[37] Michael Reeves, Dave Jing Tian, Antonio Bianchi, and Z Berkay Celik. Towards improving container security by preventing runtime escapes. In *2021 IEEE Secure Development Conference (SecDev)*, pages 38–46. IEEE, 2021.

[38] Elena Reshetova, Janne Karhunen, Thomas Nyman, and N Asokan. Security of os-level virtualization technologies. In *Nordic Conference on Secure IT Systems*, pages 77–93. Springer, 2014.

[39] Quirin Scheitle, Oliver Gasser, Theodor Nolte, Johanna Amann, Lexi Brent, Georg Carle, Ralph Holz, Thomas C Schmidt, and Matthias Wählisch. The rise of certificate transparency and its implications on the internet ecosystem. In *Proceedings of the Internet Measurement Conference 2018*, pages 343–349, 2018.

[40] Rui Shu, Xiaohui Gu, and William Enck. A study of security vulnerabilities on docker hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 269–280, 2017.

[41] Johnny So, Najmeh Miramirkhani, Michael Ferdman, and Nick Nikiforakis. Domains do change their spots: Quantifying potential abuse of residual trust. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2130–2144. IEEE, 2022.

[42] Johnny So, Najmeh Miramirkhani, Michael Ferdman, and Nick Nikiforakis. Domains Do Change Their Spots: Quantifying Potential Abuse of Residual Trust . In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (IEEE S&P)*, 2022.

[43] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. Security namespace: making linux security frameworks available to containers. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1423–1439, 2018.

[44] Samaneh Tajalizadehkhoob, Tom Van Goethem, Maciej Korczyński, Arman Noroozian, Rainer Böhme, Tyler Moore, Wouter Joosen, and Michel Van Eeten. Herding vulnerable cats: a statistical approach to disentangle joint responsibility for web security in shared hosting. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 553–567, 2017.

[45] Katrine Wist, Malene Helsem, and Danilo Gligoroski. Vulnerability analysis of 2500 docker hub images. In *Advances in Security, Networks, and Internet of Things: Proceedings from SAM'20, ICWN'20, ICOMP'20, and ESCS'20*, pages 307–327. Springer, 2021.

[46] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. What are weak links in the npm supply chain? In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 331–340, 2022.

APPENDIX A

Our experiment with re-registered domains resolved by Docker images of interest was the only experiment where we interacted with the outside world and where existing Docker containers contacted our servers. The domains were registered in order for us to be able to gauge how many deployed Docker containers are actively using the respective Docker images, by monitoring incoming connections. Had we not done that at all, we could not have made a convincing case that the affected Docker images are still in use and therefore attacks against them can have real-world implications.

Beyond registering these domains and obtaining certificates for them, we never advertised them to the outside world or attempted to attract traffic to them. Protocol-wise, the only protocol that our servers fully supported was HTTP(S) so that we could record the HTTP requests before responding with HTTP 404 messages. For all other protocols, we opened the appropriate ports that Docker clients would expect, allowed them to connect in order to log statistics about that connection, and then immediately tore it down by responding with a RST packet. Instead of failing at a resolution level (which was the case before we registered the domains in question) these connections will now fail either at a socket level (RST packets sent by server) or at the application level (HTTP 404).

Due to how we treated incoming connections, we are confident that our registration of the expired domains associated with Docker images did not negatively impact users

or systems. Connection failures before our domain-name registrations, remained connection failures after our registration. Moreover, a case could be made that the ecosystem is actually more secure as a result of our intervention. The registered domain names are now removed from the pool of available domain names and therefore cannot be abused by attackers to potentially compromise the affected Docker containers.

APPENDIX B

A. Request

Dear owner of [DockerHub project],

I believe you have a security issue in one or more of your DockerHub repositories that warrants attention.

My name is [author] and I am [author's position], conducting research in the field of cyber security. In my current project, I am seeking to understand how people use Containerized Applications (such as docker images) and how certain use cases can open them up to vulnerabilities. More specifically, I am interested in how people include remote references in these containers that could open them up to supply-chain security issues.

We discovered a potential security issue in your DockerHub repository at the following path: [DockerHub URL]

More specifically, the image itself makes HTTP requests to download ".jar" files to the [website name] website.

Depending on your use case and how you envision people using your code, this could turn into a security issue for the users who download and execute this image. That is, attackers could perform a type of MITM attack and serve a different file to the end user.

We would like to bring this to your attention and ask you if you agree with our assessment and if you would consider switching these requests to HTTPS. We are happy to provide additional information and answer any questions that you may have.

Best regards, [Author]

B. Response

Dear [Author],

Thank you for letting me know about this. I am the developer of [project name] but I am not the owner of this Docker package. This package is an unauthorized redistribution of an older version of [project name]. The current version of [project name] has been updated to make https requests instead of http requests. I have asked Dockerhub support to take down this unauthorized package.

I wish you all the best with your research project.

Best regards, [Project maintainer]

APPENDIX C

In the following tables we further investigate the pull counts of the images that belong to critically vulnerable categories, by exhibiting HTTP traffic or querying domains that yield NXDOMAIN errors. For instance, in Table VII, on our Vanilla Run, the most popular image that yielded NXDOMAIN errors had 111,544,212 pulls. Meanwhile, the overall median pull count of all vulnerable images on the same category was 94. We also note that the Vanilla Run and the Vanilla Run with our Network Scanning add-on both have the same largest pull count, as the same vulnerable image was the most popular one in both runs.

TABLE VI: *The pull counts of the images that exhibit HTTP Traffic*

Type of Run	Largest Pull Count	Top 5 Median	Overall Median
Vanilla Run	28,995,975	4,029,268	126
Vanilla Run + Net. Scan.	18,065,674	4,029,268	128
LLM-Assisted	37,415,085	898,634	222
LLM-Assisted + Net. Scan.	37,415,085	898,634	222

TABLE VII: *The pull counts of the images that attempt to connect to a domain that yields NXDOMAIN*

Type of Run	Largest Pull Count	Top 5 Median	Overall Median
Vanilla Run	111,544,212	10,870,402	94
Vanilla Run + Net. Scan.	111,544,212	13,585,725	103
LLM-Assisted	1,684,844,527	1,527,934	229
LLM-Assisted + Net. Scan.	1,684,844,527	1,527,934	229