Secrets Revealed in Container Images:
An Internet-wide Study on Occurrence and Impact

Markus Dahlmanns
dahlmanns@comsys.rwth-aachen.de
RWTH Aachen University
Germany

Constantin Sander
sander@comsys.rwth-aachen.de
RWTH Aachen University
Germany

Robin Decker
decker@comsys.rwth-aachen.de
RWTH Aachen University
Germany

Klaus Wehrle
wehrle@comsys.rwth-aachen.de
RWTH Aachen University
Germany

ABSTRACT

Containerization allows bundling applications and their dependencies into a single image. The containerization framework Docker eases the use of this concept and enables sharing images publicly, gaining high momentum. However, it can lead to users creating and sharing images that include private keys or API secrets—either by mistake or out of negligence. This leakage impairs the creator's security and that of everyone using the image. Yet, the extent of this practice and how to counteract it remains unclear.

In this paper, we analyze 337,171 images from Docker Hub and 8,076 other private registries unveiling that 8.5 % of images indeed include secrets. Specifically, we find 52,107 private keys and 3,158 leaked API secrets, both opening a large attack surface, i.e., putting authentication and confidentiality of privacy-sensitive data at stake and even allow active attacks. We further document that those leaked keys are used in the wild: While we discovered 1,060 certificates relying on compromised keys being issued by public certificate authorities, based on further active Internet measurements, we find 275,269 TLS and SSH hosts using leaked private keys for authentication. To counteract this issue, we discuss how our methodology can be used to prevent secret leakage and reuse.

CCS CONCEPTS

• **Security and privacy** → **Network security**; **Key management**.

KEYWORDS

network security, security configuration, secret leakage, container

1 INTRODUCTION

While originally developed to isolate applications [24], containerization has become a new cornerstone of interconnected services as it significantly eases their deployment [8, 54, 61, 74, 75, 79]. To this end, Docker, the most prominent containerization framework [74], uses prebuilt images that include all software dependencies necessary to deploy an application [8]. Users only need to download an image from a registry or can derive their own image by adapting its configuration and included files. These new images can then again be uploaded building a whole ecosystem of containerized applications. For example, Docker Hub, the official Docker registry, comprises more than 9,000,000 images [21] anybody can use.

With this level of public exposure, any mistake during image creation can have drastic consequences. Most notably, including confidential secrets such as cryptographic keys or API secrets, by mistake or out of negligence, can introduce two security issues: (i) attackers can misuse compromised secrets leading to potential loss of data, money, privacy, or control, and (ii) administrators instantiating images can rely on broken security, e.g., paving the way for Man-in-the-Middle attacks. Aggravatingly, there is no easy tooling to show which files have been added—accidentally adding a secret is thus much easier than identifying such an incident.

Indeed, related work traced three reused private keys authenticating 6,000 (Industrial) Internet of Things services back to the occurrence in a Docker image [12]. Additionally, blog entries produced anecdotal evidence that Docker images include further confidential security material [43, 66, 70, 77]. However, comprehensive analyses on revealed security secrets at scale do not exist in this realm. Instead, such analyses focus on GitHub repositories [33, 49, 50, 58, 62–65, 72, 88]. Hence, the extent for container images is unknown.

In this paper, we thus comprehensively study whether Docker images include confidential security material and whether administrators reuse these compromised secrets at large scale by (i) scanning publicly available Docker images for confidential security material, and (ii) measure whether these secrets are used in practice on production deployments. To this end, we analyze images available on the official and largest registry Docker Hub as well as examine the entire IPv4 address space for public registries and services relying their security on compromised secrets.
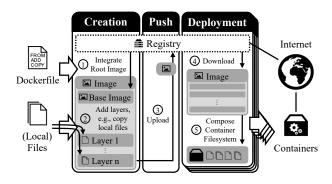
Figure 1: The Docker Paradigm: Dockerfiles describe files and functions of images. Images are uploaded to a registry for sharing and potentially numerous administrators deploy containers based on a single image (according to [8]).

Contributions: Our main contributions are as follows.

- We found 8,076 Docker registries in the IPv4 address space that contain not only secrets but also potentially confidential software and likely allow attackers to replace images, e.g., with malware.
- After filtering test secrets, we identified 55,265 leaked distinct secrets, i.e., 52,107 private keys and 3,158 API secrets, in 28,621 images (8.5 % of images we scanned are affected).
- We show that operators use 740 compromised private keys in practice affecting the authenticity of 275,269 Internet-reachable hosts providing, i.a., HTTP, AMQP, MQTT, and LDAP services.
- We discuss improvements of the Docker paradigm to prevent secret leakage and reuse in the future as well as provide our software used to find and verify secrets [9] to support mitigation.

2 A PRIMER ON THE DOCKER PARADIGM

In contrast to other containerization frameworks, Docker [24] does not only provide an isolated execution environment for applications. Instead, Docker specifies an easy-to-use paradigm to create, share and deploy ready-to-run container images [8]. These images constitute the filesystems of the containers and include all dependencies necessary for the actual applications, i.e., they can include all kinds of files added during creation. The completeness of these images allows to share them via (publicly accessible) registries. Figure 1 shows the structure and lifecycle of Docker images in detail, from creating images to sharing and running them.

Image Creation: To create an image, Docker uses a user-defined *Dockerfile* [18] to specify the image ingredients. First ①, the Dockerfile references another image, the base image, which is downloaded from a registry and comprises the initial file system of the new image. Second ②, image layers consisting of differential snapshots of the file system after running commands from the Dockerfile are created and stacked on each other [8, 20]. These commands can include shell statements to, e.g., compile an application running in the container. Furthermore, specific commands exist to embed environment variables or to add files from the host system into the image [18, 55]. While the files can be, e.g., source code or further dependencies, image creators can also easily and accidentally include (cryptographic) secrets into the image or its environment variables, putting the service's security at risk when leaked. Once an image

has been fully created, it exists as a self-containing unit, which is ready-to-run but also allows little insight on what has been added.

Image Push: After generating the image, creators can *push* it to a registry [19], e.g., the official and largest registry Docker Hub [21], allowing to deploy containers among an own fleet of servers easily, but also to share it with other users [24]. To this end, the image layers are uploaded to the registry under a repository name and tag ③. Thereby, the repository name typically represents the application in the image, and the tag describes a version. Conventionally, creators tag the newest image in a repository with latest.

Container Deployment: To run a Docker container, users *pull* an image from a registry. When pulling, users first request an *image manifest* [20] from the registry, including meta information about the image and its layers. After downloading all layers ④, Docker merges the content composing the file system for the new container ⑤ [8]. The application then finds an unchanged file system with all content provided by the image creator, i.e., all dependencies but also potentially added secrets, and can very likely provide services to the public Internet. Since numerous containers of various users can base on a single image, included, and thus compromised, secrets could affect several deployments.

Takeaway: The Docker paradigm eases distribution and deployment of applications. However, insight into what is added in images and up- or downloaded from a registry can be lost. Thus, secrets can be leaked and reused, impairing Internet-reachable services at scale.

3 RELATED WORK

Three streams of research motivate our analysis of confidential security material in Docker images: studies that detect leaked security material, research on publicly available Docker images, and Internet-wide scans disclosing security weaknesses at scale.

Actively Leaked Security Material: Currently, the search for leaked security material focuses on code repositories. Several studies detected the leakage of passwords [33], SSH private keys [49], Amazon Cloud API keys [72, 88], and Slack API keys [50], using the built-in search of GitHub. To allow broader searches, researchers entailed regular expressions but focused on specific file types [62–64] or code snippets [65], i.e., the scale of this research was limited. In contrast, Meli et al. performed a large scale study without focusing on specific file types, showing that ~3.5 % of the 4 M analyzed code repositories on GitHub included leaked secrets [58]. Further approaches use machine learning to improve the detection by relying on code semantics [28], false-positive detection [67], or both requiring further user input [45, 56]. Away from GitHub, research proposed methods to investigate various platforms [27] and proved the presence of secrets in publicly available Android apps [32]. A recent study underlines that most developers experienced secret leakage, and guidelines are insufficient for prevention [47]. While retroactively deleting leaked secrets does not help [82], (non)-commercial approaches, e.g., GitGuardian [31], TruffleHog [76], or Gitrob [37], aim at preventing secret leakage for Git.

Docker Images: Besides Git, researchers and developers, early on without evidence, assumed leaked secrets in images for virtual machines or Docker and provided countermeasures [2, 4, 5, 8, 23, 42, 80]. Nevertheless, non-academic Web-blog studies [43, 66, 70, 77] still find leaked secrets in images on Docker Hub. However, these

Table 1: Steps from our repository selection to the finally included and analyzed layers. For found repositories, we give information on their distribution over our selected search terms in a query group, and for images on their distribution over found repositories. For analyzed images, we report on the distribution of their age, and for layers, we show the distribution of their sizes. By analyzing 1,388,414 layers, we examine all files included in images of considered repositories for leaked secrets.

Query Group	Repositories (Section 4.1.1) #	distinct		Images (Section 4.1.2) #	latest	none	analyzed [+age]	Layers (Section 4.1.3) # [+size]	distinct	
Standard	480,065			7,966,650			233,649	3,967,848		
	min: 0			min: 0			min: 0 d	min: 0 B		
	p_{25}: 684	296,598	$\longrightarrow$ 79 %	p_{25}: 1	171,857	62,949	p_{25}: 409 d	p_{25}: 534 B	1,166,570	$\longrightarrow$ 76 %
	p_{50}: 3,563			p_{50}: 1			p_{50}: 899 d	p_{50}: 547 kB		
	p_{75}: 10,000			p_{75}: 3			p_{75}: 1,469 d	p_{75}: 20 MB		
	max: 10,000		357,898	max: 69,828			max: unset	max: 27 GB		1,388,414
IIoT	199,311			1,949,976			57,914	1,485,864		
	min: 0			min: 0			min: 0 d	min: 23 B		
	p_{25}: 0	76,786	$\longrightarrow$ 17 %	p_{25}: 1	44,889	18,872	p_{25}: 390 d	p_{25}: 602 B	326,298	$\longrightarrow$ 16 %
	p_{50}: 117			p_{50}: 1			p_{50}: 847 d	p_{50}: 981 kB		
	p_{75}: 7,183			p_{75}: 2			p_{75}: 1,401 d	p_{75}: 21 MB		
	max: 10,000			max: 29,357			max: unset	max: 30 GB		

studies either limit their scale [43, 70, 77] to a few thousand images/secrets or restrict their methodology [66] to process large amounts of available images. The latter study [66] finds 46,076 affected images among 6.3 M images on Docker Hub, but only considers information available in Dockerfiles, e.g., specific file paths. Meanwhile, SecretScanner [14], a smaller secret search tool, implements a function allowing users to find secrets in Docker images.

Still, a comprehensible, large-scale, and methodology-driven analysis on introduced security weaknesses by leaked security material is missing. Instead, large-scale studies on Docker images focused on data compression [86, 87], software vulnerabilities [44, 55, 84, 85], or typosquatting of image names [54]. Hence, as of now, it is unclear how widespread secret leakage is in images on Docker Hub as well as private Internet-reachable registries. Moreover, it is unknown to what extent these compromised images are then used on the Internet and whether they weaken security at scale.

Internet Measurements: For understanding deployment security at scale, Internet-wide measurements have been a valuable tool in the past. Internet scan services, such as Shodan [71] or Censys [25], fetch and publish meta-information, e.g., security configurations, on Internet-reachable services. Although these services often helped researchers analyzing the security of connected devices, e.g., cars [78] or (insecure) Industrial IoT (IIoT) deployments [35], they usually do not see all deployments [3]. Hence, researchers frequently conduct own active Internet measurement, e.g., using ZMap [26]. On the web, these measurements allowed to analyze the deployment of new TLS versions [41, 51] and revealed wide security configuration mistakes [7, 10, 38–40, 48, 52] or implementation deficits [1, 36, 73]. Aside the web, researchers assessed the security of SSH services [29, 81] and key-value stores leaking confidential data [34]. For the IoT and IIoT, research revealed many deployments relying on vulnerable software [6, 30, 46] and communicating without any security mechanism [12, 57, 59, 60, 83], e.g., access control. Even with built-in security features, operators often configure such services insecurely [11]. For example, a massive reuse of certificates was traced back to a Docker image including certificates and corresponding private keys [12] jeopardizing the authenticity of numerous deployments. Based on this, we claim that it is probable that there are further public Docker images that wrongly include

confidential secrets and harm security on the Internet—especially when looking at the sheer size of Docker and Docker Hub.

Takeaway: *Although the broad leakage of security secrets in code repositories is well understood, the spread of revealed secrets in Docker images and the introduced security risk for the Internet are unknown. However, known secret leakage detection techniques and Internet measurements are predestined to shed light on these issues.*

4 COMPOSING OUR DATASET

To answer whether Docker image creators actively compromise security secrets by publishing them in openly available Docker images, we set out and retrieve images from Docker Hub (Section 4.1) and publicly reachable private registries (Section 4.2).

4.1 Retrieving Images from Docker Hub

Table 1 guides through our composition process on Docker Hub, which has three tasks: (i) composing a list of *repositories*, (ii) selecting one *image* per repository to widely spread our analysis, and (iii) identifying *layers* the images consist of.

4.1.1 Repositories. While Docker Hub limits the number of image downloads [22] and we cannot download and analyze all 15 PB of images available on Docker Hub [13] due to runtime and bandwidth restrictions, our analysis requires a selection of repositories of interest. Furthermore, Docker Hub does not support listing all available images to choose from. Hence, we use specific search terms to get images users retrieve when searching via the Web interface. Our search terms (which we elaborate in more detail in Appendix B) build two query groups (Table 1 (left)); **Standard** comprises mainstream communication protocol names [68] and frequently used technologies [74] for a wide analysis of images referencing current issues. For comparison and more focusing on a specific area, we choose the Industrial Internet of Things (IIoT) as past studies showed a great susceptibility to security faults [11, 12, 35, 53, 59, 60], i.e., **IIoT** includes protocol names from this area.

We list the number of repositories covered by our analysis per query group, i.e., the sum of found repositories of all search terms of a group, in Table 1 (column Repositories-#). To further convey

Table 2: Overview of found private Docker registries, available image repositories, their tags, layers, and final layers included in our dataset. We added 258,889 randomly selected image layers, preferably from images tagged with `latest`, to our dataset.

Date	# Registries	Repositories #	distinct	Images #	latest	none	selected	Layers #	distinct	selected [+size]
22-08-01	8,076 non-TLS: 7,593 TLS: 483	81,570 min: 1 avg: 10 max: 100	51,163	6,160,615 min: 0 avg: 75 max: 38,638	34,377	3,704	55,746	2,480,137	516,913	258,889
22-08-06	5,656 non-TLS: 5,184 TLS: 472	77,900 min: 1 avg: 13 max: 100	50,611	6,350,311 min: 0 avg: 81 max: 38,638	31,997	3,289	55,071	2,445,347	491,085	551,617 min: 87 B avg: 6 MB max: 249 MB

(distinct repositories overall: 53,448)

the prevalence of our search terms, we indicate the minimum, maximum, and 25-, 50-, and 75-percentiles of search results for included terms, i.e., higher values of lower percentiles would imply a higher prevalence. While both query groups contain terms that lead to no results (*min*), i.e., the term is not mentioned in any repository name or description, terms in the standard group generate more results due to their closer correlation to frequently used technologies than IIoT protocols (p_{25}, p_{50}, p_{75}). Docker Hub's API limits the number of results to 10,000 (*max*).

As different search terms lead to overlapping repositories, we further report on the distinct number of repositories gradually, i.e., per query group, and overall. In total, we gathered 357,898 distinct repositories subject to our study of which 79 % are uniquely added by our standard search terms and 17 % by IIoT related search queries.

4.1.2 Images. Table 1 (column Images-#) indicates how many images were available in total over the distinct repositories of a search group. While repositories mostly contain different images, including the same software in other versions and thereby comprising similar files, we choose to analyze one tag per repository to spread our analysis as widely as possible. Here, we select images tagged with `latest` which is used as Docker's default and typically includes the newest version of an image. However, not all repositories contain images tagged with `latest` (as shown in Table 1 (column Images-`latest`)). Here, we select the image with the latest changes (as reported by Docker Hub's API). Empty repositories (Table 1 (column Images-none)), i.e., have no image layers available, cannot include any secrets. Besides the number of images that are covered by our study (column Images-analyzed), we also report on the age of the images to analyze how long they are already available on Docker Hub. The ages of images included in both query groups roughly have the same distribution indicating that although the number of images found by our IIoT-related queries is lower image creators update their images in the same frequency as image creators of images included in our Standard group.

4.1.3 Layers. While we report on the number of layers included in all images (Table 1 column Layers-#), different images often share the same layers, e.g., layers from frequently used base images. Hence, to speed up our search for leaked secrets, we analyze each distinct layer only once. We show the distinct number of layers gradually, i.e., per query group, and overall. To cover all 357,898 repositories, we analyze 1,388,414 layers. (76 % uniquely added by Standard-related, 16 % by IIoT-related repositories).

4.2 Images from Private Docker Registries

Since image creators might upload sensitive images preferably to private registries, we want to include images from these registries in our analysis. Table 2 shows our steps taken to extend our dataset with images from private registries, i.e., we search private registries, and, subsequently, include a subset of available layers.

4.2.1 Find Private Registries and Repositories. To find publicly reachable Docker registries, we scan the complete IPv4 address space for services running on the standard port for Docker registries, i.e., TCP port 5000, under comprehensive ethical measures (cf. Appendix A) twice to analyze short-term fluctuations (Table 2 (left)). Both times, we perform a TCP SYN scan using zmap [26], identifying hosts running a service behind this port and subsequently send an HTTP request as defined by Docker's Registry API [19] for verification. Whenever we do not receive a valid HTTP response, we retry via HTTPS. While we found up to 8,076 private registries on 22-08-01, the difference in found registries in comparison to our scan on 22-08-06 is due to registries in Amazon AWS-related ASes that do not reply after our first scan anymore. Since these registries only contain the same and single image (uhttpd), they might relate to another research project, e.g., implementing a registry honeypot.

Contrarily to Docker Hub's API, the API of private registries allows listing available repositories without search terms. However, we limit our requests to receive a maximum of 100 repositories per registry to prevent any overloads. As such, the found private registries provide 81,570 resp. 77,900 repositories. Since the registries do not implement access control for read access, clients are able to download all included images. Notably, by default also write access is not restricted [17], i.e., attackers might be able to inject malware.

While being publicly available on private registries but not filtered by any search terms, the content of these images is of special interest. Here, often the repository name indicates the image's content and thus allows conclusions on widely distributed applications, i.e., over both measurements, uhttpd is the most reoccurring repository name (reoccurring 2,596 times, but only during our first scan). Repository names on the second and third place, i.e., nginx and redis, indicate proxy and cloud services where image creators might have included security secrets before uploading it to their registry. Beyond the scope of security secrets, other repository names occurring less often, e.g., api-payments-gateway_prod or smarthome_web, imply that image creators might include confidential software, source code, private data, or information on systems especially worthy of protection in openly available Docker images.

Table 3: Domains of secrets covered in our analysis, their potential threats (left), the number of (distinct) matches we found with our corresponding regular expressions in images on Docker Hub and private registries (center), as well as the number of matches we validated (right). We excluded rules with too arbitrary, thus unverifiable, matches (gray) from validation process.

		Regular Expressions *(Section 5.1.1 / Appendix C)*	(Distinct) Matches *(Sec. 5.1.2)*		Valid Secrets *(Section 5.1.3)*		
	Domain	Potential Threat / *(Service) Type*	Images	Variables	Images	Variables	Total
	Private Key	Perform man-in-the-middle attacks, fake identity, …	1,377,336	2	52,107	0	52,107
		PEM Private Key, PEM Private Key Block, PEM PKCS7, XML Private Key	(62,282)	(1)			
API	Cloud	Manage services, create new API keys, reconfigure DNS, access emails / SMS, control voice calls, read / alter private repositories, …	6,208,995	416	2,880	67	2,920
		Alibaba[76], Amazon AWS[76], Azure[76], DigitalOcean[76], Github[76], Gitlab (v1, v2)[76], Google Cloud[76], Google Services[58], Heroku[76], IBM Cloud Identity Service[76], Login Radius[76], MailChimp[58], MailGun[58], Microsoft Teams[76], Netlify[76], Twilio[58]	(74,460)	(84)			
	Financial	List / perform payments, inspect / alter invoices, …	42,901	4	23	2	25
		Amazon MWS[58], Bitfinex[76], Coinbase[76], Currency Cloud[76], Paydirt[76], Paymo[76], Paymongo[76], Paypal Braintree[58], Picatic[58], Stripe[58], Square[58], Ticketmaster[76], WePay[76]	(543)	(2)			
	Social Media	Tweet, access direct messages, retrieve relationships, …	6,365,854	14	209	4	213
		Facebook[76], [58], Twitter[58]	(439,822)	(8)			
	IoT	Retrieve (privacy-sensitive) IoT data, e.g., track cars, …	297	0	0	0	0
		Accuweather[76], Adafruit IO[76], OpenUV[76], Tomtom[76]	(117)	(0)			

4.2.2 Image and Layer Selection. For all found repositories, we collect the lists of available images and their tags (Table 2 (center)). Although private registries typically do not implement any rate limiting like Docker Hub, we do not want to overload found registries or their Internet connections. Hence, to spread our analysis as far as possible but limit the load on each registry, we choose one tag per image. Similar to our selection process on Docker Hub, typically, in each repository, we select images tagged as `latest` to download the corresponding manifest. Whenever no `latest` image is available, we sort all available images naturally by their tag (to account for version numbers as tags), and select the maximum (i.e., the newest version), as the API does not provide any information on the latest changes. Subsequently, we download the corresponding image manifests to retrieve accompanying layers. To further limit load on Internet connections of found registries, we do not download all available layers for included secrets. Instead, we randomly select layers of chosen images such that the sum of their sizes does not exceed 250 MB per registry and per measurement. All in all, we added 258,889 layers from private registries to our dataset.

Takeaway: In parallel to Docker Hub numerous private registries exist providing images to the public. Overall, we assemble a dataset of 1,647,300 layers from 337,171 images subject to our future research. Furthermore, private registries might allow attackers to, e.g., inject malware, potentially infecting container deployments at scale as well.

5 LEAKED SECRETS IN DOCKER IMAGES

Next, we search in considered images for included secrets (Section 5.1), discuss the origin of affected images to later evaluate remedies (Section 5.2), and analyze also found certificates compromised due to private key leakage to estimate arising risks (Section 5.3).

5.1 Searching for Secrets

To analyze available images for included secrets, we align our approach to established methods [58, 76], i.e., we choose and extend regular expressions identifying specific secrets and match these on files and environment variables. Additionally, we extensively filter our matches to exclude false positives.

5.1.1 Regular Expression Selection. We base our selection of regular expressions on previous work to find secrets in code repositories [58, 76] (we further elaborate on our election process and expressions in Appendix C). Table 3 (left) names the domains of secrets that our selected expressions match and indicates how attackers could misuse these secrets. We start with regular expressions composed by Meli et al. [58] due to their selection of unambiguous expressions (reducing false positives) matching secrets with a high threat when leaked. We extend their expressions for private keys to match a larger variety, e.g., also OpenSSH private keys. Moreover, we widen the set by expressions matching API secrets of trending technologies [74] based on match rules from Truffle-Hog [76]. However, TruffleHog's rules are relatively ambiguous and incur many false positives, which TruffleHog filters by validating the API secrets against their respective endpoints. As our ethical considerations do not allow for any further use of the secrets (cf. Appendix A), we focus on rules which expect at least one fixed character and later add further filtering and verification steps.

5.1.2 Matching Potential Secrets. To analyze whether image layers include secrets, we match the selected regular expressions on the images as follows (we will open-source our tool on acceptance of this paper): We download and decompress the image layers and then match our regular expressions on the included files. Moreover, we recursively extract archive files up to a depth of 3 and match again. As API documentations often suggest setting secrets in environment variables and not writing them into files, we analyze set variables. Since Docker allows downloading the small image configuration containing set variables aside of the image, i.e., potential attackers do not have to download and search through all files to find included secrets, we analyze variables separately: As such, we only download the image configuration file and iterate our regular expression over set environment variables. Here, we adapt the API expressions, as some expect a specific term before the secret (cf. Table 5 in Appendix C), e.g., the service name as part of a variable name. As the variable names and values are separated in the configuration file, we also split the according expressions and match them individually.

Table 3 (center) lists for each secret domain how many matches and how many distinct matches we found in both, image content and environment variables. Notably, while only covering two services, i.e., Facebook and Twitter, the expressions in the Social Media domain matched most often over all domains, which already indicates that API secrets of this domain are often suspect to leakage.

The high redundancy of the matches, visible as the significant decrement between distinct and non-distinct matches, already hints at invalid matches, e.g., private keys or example API tokens prevalent in unit tests or documentation in several layers. Indeed, the most reoccurring match `AKIAIOSFODNN7EXAMPLE` (291,949 times in 20,497 different layers), is an example key for Amazon AWS API from a library documentation which creators usually include in their images. We thus validate our matches extensively.

5.1.3 Match Validation. To exclude test keys for cryptographic libraries, example API secrets, and completely invalid matches to get a near lower bound of harmful leaked secrets in Docker images, we use different filters depending on the secret type. While we show the number of resulting valid secrets in Table 3 (right), Figure 2 details the filtering results separated by the match's origin, i.e., image content or environment variable and domain.

Private Keys: Our regular expressions for private keys match on PEM or XML formatted keys. Thus, we can first exclude every match that is not parsable (filter *Unparsable*). Figure 2 shows that only a minority of all potential private keys in image layers are unparsable, underlining that image creators include and compromise private keys actually usable in final Docker containers for practical operations. Contrarily, the single match within the environment variables is only a key fragment and thus not parsable.

Still, we expect a high number of software test keys in Docker images among found keys, as they are part of several libraries creators might include in their images, e.g., OpenSSL. Since users will most likely not use such keys to secure their deployments, we filter out test keys that are included in kompromat [69], a repository listing already compromised secrets (filter *Kompromat*). More specifically, we filter keys occurring in RFCs (6), libraries for software tests (1,820), or as special test vectors (3).

To also account for software test keys that are not available in kompromat, we analyze the file paths where respective keys were found (filter *File*). While we do not generally exclude all paths containing signal words indicating test or example keys, as users might use such paths also for keys they generated and use in practice, we apply different measures. For instance, based on locations of test keys identified using kompromat, we deliberately exclude matches in similar locations, i.e., keys within directories where we already detected test keys and all parent directories under which we find more than 2/3 test keys. Last, we exclude file paths typically used by libraries (cf. Appendix D), e.g., `/var/lib/*`, as there is a lower chance that users adapt their keys here.

Figure 2 shows that these filters process the largest share of excluded private key matches. It further indicates that kompromat only includes a minority of software test keys, i.e., is not directly usable to exclude all false-positive matches. Still, many of the found keys are not filtered and, thus, most likely, no software test keys.

In total, we found 52,107 valid private keys potentially in use in practice (cf. Table 3 (right)). Since all of these keys are located in
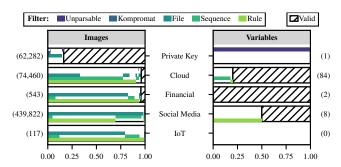

Figure 2: Validation of matches. While most private key matches are valid, API secret matching in Docker images is challenging and requires well-configured filters. Several filters treat a large share of matches in parallel. Absolute number of distinct matches in parentheses.

files, attackers would have to download respective image layers to get access and not only meta information to retrieve environment variables. Still, since these keys are publicly available and thus compromised, usage in production puts authentication at stake, i.e., attackers can perform impersonation attacks.

API Secrets: Since our ethical considerations deter us from validating API secrets against their service endpoints (cf. Appendix A) as applied by TruffleHog [76], and related methods for false positive detection focus on matches in source code [28, 45, 56, 67], which is not prevalent in Docker images, we need alternative measures to filter invalid matches. By manually supervising our filtering, we ensure that the final set only includes valid-looking API secrets.

Based on invalid matches in GitHub code repositories [58], we expect human-created example keys that contain keywords, e.g., `EXAMPLE`, or consecutive character sequences, e.g., `XXXX`, that we must exclude (filter *Sequence*). To filter consecutive sequences, we search for segments consisting of ascending, descending (both with a length of four), and repeating characters (with a length of three). Furthermore, we filter matches including sequences that occur unusually often, i.e., we create (4, 7)-character-grams of all matches, exclude grams created over fixed parts of our regular expressions as well as grams only containing digits, and count the number of occurrences over all API matches. To account for randomly reoccurring grams, we filter all matches that include grams occurring 29 times more often than the average. We manually ensured that our filter is not too restrictive but also not to loose leaving often reoccurring grams out. Figure 2 shows that this filtering excludes a large share of matches. Interestingly, the most reoccurring gram is `. . . .` [sic!], which we could trace back to DNA sequences in images related to bioinformatics underpinning the large variety of different and unexpected file types occurring in Docker images.

Similar to filtering private key matches by their file paths, we also filter API matches occurring in manually selected paths (filter *File*, cf. Appendix D). Essentially, we revisited the location and file types of all matches and excluded paths that most likely do not include any valid secrets compromised by publishing these in Docker images. Figure 2 indicates that the filtered paths often also include matches filtered by our sequence filter and thus that libraries include strings similar to secrets, e.g., in their documentation.

Still, after manual revision of the remaining matches, we conclude that rules which match on a fixed term before the secret, e.g.,
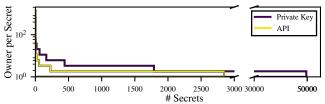
Figure 3: Total secrets found in images of a specific number of owners. Most were found in images of single owners.

the service name, and then allow a specific length of characters are too ambiguous for usage on files in Docker images as they match on arbitrary content, e.g., on hashes with the service name in front. We thus decide to exclude matches of these rules from our further analysis (gray in Table 3 (left)), i.e., consider these matches invalid, to ensure the integrity of our further results. Still, a minority of these matches might be valid, potentially enabling attackers to compromise production services or access confidential data.

Comparing the filter results of API secret matches in files and environment variables, the share of valid matches in variables is significantly higher than in files indicating that image creators less likely include secret placeholders in variables. Still, as Table 3 (right) shows, most secrets are located within the images. Thus, attackers have a higher chance of finding valid secrets when downloading both environment variables and image content.

In total, we found 3,158 distinct API secrets in Docker images, mostly related to services from the cloud domain (2920 secrets). Although we cannot prove the functionality of these secrets, the occurrence of 1,213 secrets for the Amazon AWS API or 177 secrets for the Alibaba API indicate that attackers might be able to reconfigure cloud services maliciously, e.g., by editing DNS or VM options. Additionally, we found evidence for secrets allowing attackers to access private data from social media (213 secrets), or even access financial services (25 secrets, most matches: Stripe API). Notably, although we focused our image search partly on IoT terms, we found no valid secrets from selected IoT services.

5.1.4 Secrets Owned by Single Users. Based on findings over leaked secrets found on GitHub [58], we expect most valid secrets to residing in images of single users (as users do not share their secrets intentionally). Contrarily, invalid matches, e.g., library test keys, would mainly reside in images of multiple owners.

Thus, to check whether the matches we identified as valid secrets are located in images of single users, we analyze the number of different owners that include a specific secret in their images. To this end, for images from Docker Hub, we consider the repository owner (embedded in the repository name) as the owner of a secret. For private registries, we consider the registry's IP address as the owner (assuming that owners only run a single registry and neglecting that registries might use different (dynamic) IP addresses).

Figure 3 shows that the largest share of valid secrets indeed occurs in images of single owners. 95 % of private keys (49,667 keys) and 90 % of API secrets (2,845 secrets) reside in images of single owners underpinning that these should be protected. Moreover, we can trace 95 private keys and 19 API secrets of multiple owners back to inheritance. These secrets were already included in the base image, but w.r.t. to the overall occurrence, we conclude that secret spread due to inheritance is no major problem.

To responsibly inform image creators about leaked secrets in their images, we reach out to them whenever possible (1,181 extractable and valid e-mail addresses) and also contacted the operator of Docker Hub (cf. Appendix A). Early on, we received notifications of creators that removed found secrets from their images.

Takeaway: *55,265 found secrets show that image creators publish confidential information in their publicly available Docker images. As attackers have access to these secrets relying authentication and other security mechanisms are futile, potentially leading to compromised servers or leaked privacy-sensitive data.*

5.2 Origin of Leaked Secrets

Next, we analyze where the validated secrets stem from to see whether specific images are more affected and why. To this end, we examine the distribution of affected images and compare between private registries and Docker Hub, as well as IIoT specific and Standard images. Moreover, we evaluate which operation in the original Dockerfile led to the insertion of secrets and inspect the file paths where they reside to get an intuition for their usage.

5.2.1 Docker Hub Leads Before Private Registries. We already discovered that private registries include potentially sensitive images. However, until now, it remains unclear whether images on these registries are more often subject to secret leakage than images from Docker Hub, e.g., due to creators believing that these are unavailable for the public. Thus, we analyze whether leaked secrets occur more often in images from Docker Hub or from private registries.

While we found that 28,621 images (8.5 % of images analyzed) contain valid secrets, 9.0 % of images from Docker Hub and 6.3 % of images from private registries are affected. Thus, creators upload secrets to Docker Hub more often than to private registries indicating that private registry users may have a better security understanding, maybe due to a deeper technical understanding required for hosting a registry. Yet, both categories are far from being leak-free.

For Docker Hub, besides the increased fraction of leaked secrets, we see an issue for others, i.e., other users can easily deploy containers based on these images. Thus, there is a higher chance their containers rely their security on included and compromised secrets. For example, a shared certificate private key could lead to an impersonation attack. In case of shared API secrets, all deployed containers might use the same API token leading to exhausted rate limits in the best case, but maybe also to overwritten or insufficiently secured private data. As a single API token does not allow fine-granular exclusions, i.e., it is either valid or revoked for all users, a revocation would also interfere with benign users.

Independent of their origin, attackers could equally misuse the secrets we found to leverage authentication or access privacy- or security-sensitive data. As such, both user groups of Docker Hub and private registries leak sensitive information, be it through unawareness or a deceptive feeling of security.

5.2.2 Domains are Similarly Affected. For our image selection on Docker Hub, we specifically included search terms relating to the IIoT, as past research has shown significant security shortcomings in this area. However, until now it is open whether found images of a certain domain are suspect to revealed secrets more frequently
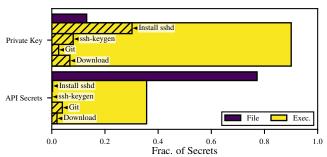
Figure 4: Operations that include secrets. Most API secrets are inserted by file and private keys by execution commands.

than other images. To answer this question, we trace images that include secrets back to the query group that led to their inclusion.

We discovered that 7.2 % of the images only found using queries from the Standard query group and 6.2 % of images only from the IIoT group include valid secrets[1]. Thus, in case of secret leakage via Docker images and based on our selected search terms, the IIoT domain does not perform worse than our Standard domain. However, it underpins that the problem of secret leakage in Docker images is a prominent issue for all domains.

5.2.3 Fresh Private Keys and Copied API Secrets. To find countermeasures against secret leakage in Docker images, it is important to understand how these leaked secrets became part of Docker images. More specifically, for private keys, it is unclear whether creators execute commands in the Dockerfile to create fresh keys, which are then published in images, or whether they manually add them, i.e., using ADD or COPY in a Dockerfile. Additionally, both, private keys and API secrets, could be indirectly included through other means, e.g., by cloning Git repositories or downloading further data.

Figure 4 shows that while most API secrets are typically inserted by file operations (*File*), e.g., copied from the image creator's host system, private keys are predominantly included by executing a command within the Dockerfile (*Exec.*)[2]. Thus, private keys might be either downloaded or generated during the creation process.

To further trace the insertion of secrets in Exec. layers back to the responsible executed commands, we analyze these commands. Since image creators often concatenate several bash commands whose output is then included in a single layer without any opportunity to associate files (and thus secrets) to a specific command, we count each of the commands related to the leakage of a secret. We show the most prominent of all 575 commands associated with secret leakage in Figure 4. In fact, 30 % of private keys were generated in layers where image creators installed the OpenSSH server. Since the installation triggers ssh-keygen to generate a fresh host key pair, it is automatically included in the image.

While the procedure of automatic key generation is beneficial on real hardware, i.e., users are not tempted to reuse keys on different hosts, in published Docker images it automatically leads to compromised keys and thus puts the authenticity of all containers relying on this image in danger. Further 8.1 % of found private keys were generated by a direct call of ssh-keygen, e.g., to generate fresh

---

[1]Images found by both query groups are not included.
[2]Secrets can be associated with both, File and Exec. operations, e.g., when first ADDed to the image and then copied or moved internally using cp or mv.

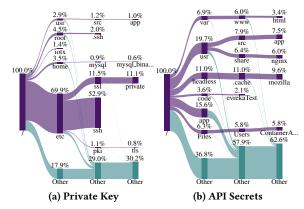(a) Private Key   (b) API Secrets

Figure 5: Most frequent file paths where we found secrets. While the major location of found private keys focuses on a few file paths, i.e., most private keys are stored where SSH host keys reside, API secrets are spread further.

SSH client key material, implying the planned usage in production of generated but compromised key material.

Given the massive secret leakage on GitHub [58], we also expect secrets to be included in images by cloning Git repositories. However, only a minority of secrets can be associated with Git, suggesting that the sets of users leaking secrets via Docker and GitHub are distinct. Furthermore, only a minority of secrets were downloaded (using wget or curl) both indicating that the secrets we found were most likely exclusively leaked in Docker images and underpinning that they are actually worth being protected.

5.2.4 File Paths Indicate Usage. To further reason about the usage of our found secrets, we analyze their file paths within the images assessing where secrets stem from and how services apply them. Separated by private keys and API secrets, Figure 5 shows the distribution of secrets throughout the directory structure of all images and focuses on the top seven paths.

We found the majority of private keys in /etc/ssh underpinning a high prevalence of compromised SSH host keys. Another large share occurs in /etc/ssl suggesting compromised keys used for host authentication via TLS. This path is also the location for TLS default (*"snakeoil"*) keys that are used if no other information is provided. They are auto-generated when the ssl-cert package is installed such that every host possesses a unique default key-pair. However, when installed during the creation of Docker images, the key is included in the image and, thus, compromised when shared. Based on the key's filename, indeed, we found 6,447 of such keys which are potentially used to offer TLS services with broken authenticity to the public Internet.

Even more alarming, we found keys lying in /etc/pki, indicating that included keys are associated with a Public Key Infrastructure (PKI), and thus potentially destined to offer services to a higher number of users. Furthermore, /iotx contains private keys used in relation to the IoT and, as per the repository names, for authentication using IoT protocols like CoAP and MQTT. Thus, attackers possessing these private keys can leverage the authentication of all connections users establish to each container created based on these images. In fact, attackers then can access or alter transmitted confidential information, e.g., privacy-sensitive user data or

commands of IoT services potentially impacting cyber-physical systems. In addition, we found keys in `/root/.ssh`, i.e., a location where SSH client key pairs typically reside. Hence, these keys might enable attackers to take over SSH servers, trusting these keys and having access to confidential data.

Contrarily, found API secrets are distributed more evenly through the directory structure. We found the largest share in `/app`, which is the example folder for including own applications in Docker images [16], underlining that image creators compromise their own application's API secrets. While similar holds for `/usr/src/app`, another large share of secrets resides in `/headless/.cache/mozilla` stemming from Firefox profiles containing Google Service API secrets in cached JavaScript files. Although these secrets are most likely usable in combination with Google Maps or Google Analytics and thus meant to be shared with website visitors, this leakage implies privacy issues: An attacker could retrace the creator's browsing history, which apparently exists due to the cache being filled, which could show potentially sensitive information.

In addition, we found a large share of Google API secrets (both Cloud and Services) in `/code/evrekaTest`. Since we do not use API tokens for further validation (cf. Appendix A), we cannot be entirely sure whether these secrets are usable or only generated for testing purposes. However, manual supervision of the matches and including files suggest that they could be actually in use.

Takeaway: *8.5 % of analyzed images contain and thus leak secrets. While the majority stems from public Docker Hub images regardless of their domain, also private registries leak a significant number of secrets. Notably, associated file paths and commands imply their production use and that various authentication mechanisms are futile.*

5.3 Compromised Certificates

To further understand the severity of potentially compromised systems, we now focus on found certificates as they provide various information on their relations and use cases. Thus, we research the trust chain, validity, and usage parameters of 22,082 compromised certificates occurring in Docker images.

Trust Anchors: While self-signed certificates indicate the usage of certificates in controlled environments, i.e., clients need a safelist with all certificates they can trust, CA-signed certificates imply the usage at larger scale as these are trusted by all clients having a corresponding root certificate installed. We consider certificates where the issuer and common name are similar as self-signed and CA-signed otherwise. For CA-signed certificates, we consider those which we can validate against widespread root stores[3] as signed by a public CA, and otherwise signed by a private CA.

We discovered that the majority of found compromised certificates (61 %) are self-signed, but also 7,546 private CA-signed and 1,060 public CA-signed certificates. While all systems relying on these certificates open the door for impersonation attacks, the occurrence of CA-signed certificates is especially alarming as such certificates are typically planned to provide authenticity to many clients/users and are universally accepted. Thus, knowing these certificates' private key not only allows attackers to perform Man-in-the-Middle attacks but also enable them to sign malicious software to compromise other's systems.

---

[3] Stores from Android, iOS/MacOS, Mozilla NSS, OpenJDK, Oracle JDK, and Windows.

Validity: As a countermeasure against key leakage, the certificate's lifetime enforces service operators to request new certificates from time to time, as clients should reject outdated certificates. Notably, 141 public-CA, 4,970 private-CA, and 10,629 self-signed certificates were valid when we downloaded their containing image layer, showing that the authenticity of relying services is at stake, i.e., the lifetime does not help in these cases of key leakage.

Interestingly, 631 public-CA, 6,486 private-CA, and 12,263 self-signed certificates were valid when added to their Docker image (as per the image's history timestamp). While these larger numbers show that the limited lifetime of certificates helps to mitigate leaked private keys, they also indicate that key leakage in images is tedious, i.e., more and more private keys are leaked.

Usages: The usage attributes of certificates can optionally indicate the practical use-case of CA-signed certificates and, thus, further help to understand the severity of the private key leakage. While all public-CA-signed certificates allow for authentication (digital signatures), and 799 are explicitly declared for server authentication, 3 (private-CA: 22) allow for code-signing. Thus, knowing the private key of these certificates, does not only allow attackers to perform Man-in-the-Middle attacks, but also enable to sign malicious software to compromise others systems.

Takeaway: *22,082 found compromised certificates show that leaked private keys can have extensive influence on the authenticity of services and software. Thus, attackers can impersonate services, decrypt past communications, or sign malware to infect production systems.*

6 SECRET USAGE IN THE WILD

Until now, it is open whether the found compromised secrets are used in practice and, if so, to what extent, i.e., whether a single compromised secret is reused due to several Docker containers stemming from the same image. While we cannot check the validity of API secrets by using them against their destined endpoint due to our ethical guidelines (cf. Appendix A), we can investigate whether hosts on the Internet use found private keys for authentication.

To assess whether Internet-reachable hosts can be suspect to impersonation attacks due to secret leakage in Docker images, we check for TLS- and SSH-enabled hosts relying their authentication on compromised private keys by using the Censys database, i.e., 15 months of active Internet-wide measurement results [25]. Here, we search for hosts presenting a public key, i.e., as SSH host key or within a TLS certificate, matching to one of the found compromised keys. More specifically, we match the fingerprint of public keys in the Censys database on ones extracted from found private keys.

In Figure 6, we detail how many hosts rely their authenticity on found compromised private keys and how often these keys are reused. While the total number of hosts relying on compromised keys is worrying on its own (275,269 hosts in Oct. 2022), their protocols, even worse, imply sensitive services. As such, in October 2022, we find 8,674 MQTT and 19 AMQP hosts, potentially transferring privacy-sensitive ((I)IoT) data. Moreover, 6,672 FTP, 426 PostgreSQL, 3 Elasticsearch, and 3 MySQL instances serve potentially confidential data. Regarding Internet communications, we see 216 SIP hosts used for telephony as well as 8,165 SMTP, 1,516 POP3, and 1,798 IMAP servers used for email. Since these hosts are susceptible to impersonation attacks due to their leaked
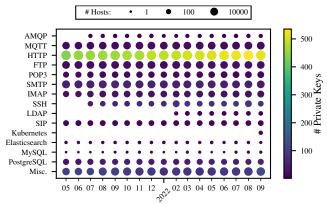
Figure 6: Alarming number of hosts (dot size) relying their authenticity on several compromised keys (dot color) over time (x-axis). Used protocols (y-axis) imply sensitive services.

private keys, attackers can eavesdrop, relay, or alter the sensitive data transmitted here.

Aggravatingly, we also find services with administrative relevance: 240 SSH servers rely on 77 compromised host keys and 24 Kubernetes instances use leaked keys opening doors for attacks which can lead to remote-shell access, extension of botnets or further data access. The comparably low number of compromised keys used (compared to 11,942 found SSH host keys) is probably due to a missing need for SSH servers in Docker containers as other mechanisms, e.g., `docker exec`, already allow shell access. Furthermore, we see 37 LDAP instances relying on leaked secrets. As LDAP is used as a base for user authentication on attached systems, the integrity of unknown many other clients is at stake. For instance, attackers could grant themselves root access to a myriad of systems.

The number of actually used keys is low compared to the number of hosts which rely on them indicating that a few Docker images lead to various compromised container deployments. Thus, the simplicity of Docker to deploy services based on ready-to-use images puts the authenticity of several instances most likely operated by different users under threat. In this regard, HTTPS hosts stand out in particular. 213,573 HTTPS hosts use 515 different compromised private keys showing that the reuse of these keys is rampant for Web services. Thus, attackers can perform Man-in-the-Middle attacks to alter webpages on their delivery or data sent to the server.

Figure 6 also underpins that the key usage of compromised keys is long-lasting and rising, i.e., over the complete available period the number of compromised systems grew from 243,419 (relying on 535 compromised keys) to 275,269 hosts (740 keys) indicating that container images with compromised certificates or SSH host keys included are increasingly used. Thus, the authenticity of more and more systems is futile, offering an ever-growing attack surface.

While our study is significantly driven by initially found compromised keys in Docker images in the area of the IIoT, Censys does not identify secured IIoT protocols other than AMQP and MQTT via TLS. Thus, we perform own Internet-wide measurements for a deeper inspection of whether IIoT services also use compromised certificates, e.g., for authentic communication via OPC UA. To this end, we select ten secure IIoT protocols from recent literature [12] and mimic its proposed measurement strategy. Our results show

that besides the already large number of compromised AMQP and MQTT hosts, only 2 CoAP hosts use 2 different leaked keys from Docker containers. That we do not find substantially more compromised hosts using other IIoT protocols underlines that the issue of key leakage is not an IIoT specfic hotspot but a general problem.

Takeaway: *275,269 hosts use 740 compromised private keys found in Docker images for authentication on the Internet and encompass deployments using, i.a., MQTT, SMTP, and PostgreSQL. This widespread usage allows attackers to eavesdrop on confidential or alter sensitive information, e.g., from the IoT, webpages, or databases.*

7 DISCUSSION, LIMITATIONS & MITIGATIONS

The outcome of our work has different aspects. We have seen that numerous private keys are compromised by image creators publishing their images via Docker registries and shown that security relies on these secrets in practice. Still, future work could investigate the *limitations* of our approach or implement the derived *mitigation opportunities* from our results.

View on Available Images: Due to rate and computation-time limits and comprehensive ethical considerations (cf. Appendix A), we could not analyze all available images on Docker Hub and private registries. Thus, we might have missed secrets included in single layers or complete images that were not subject to our study. In this light, the absolute number of found secrets is already very alerting. Also, in relative numbers, our results should be representative for the selected groups due to our sampling. Yet, the selected groups, i.e., our Docker Hub search terms, might lead to skewed results overestimating the overall population. For instance, images that are not targeted at protocols might have been created with fewer secrets. Thus, we opted for a broad body of terms based on, i.a., public polls [74] to avoid any bias. Moreover, our private registry analysis has not been targeted but included randomly sampled layers, and we still found a similar share of affected images as on Docker Hub. As such, we believe that our relative results are—at least in their magnitude—representative for the overall population of Docker images publicly available.

Missing Methods to Check API Secrets: While relying on Internet-wide measurements was a suitable measure to assess the usage of compromised private keys for the authenticity of Internet-reachable services, we could not check whether found API secrets are functional. The only option would be to contact the corresponding API's endpoint to check for the acceptance of found credentials. However, due to our ethical considerations, we must not use found secrets as such usage might influence other systems or services. Thus, we cannot validate them against their respective endpoint. Still, the number of found secrets is worrying and looking at the usage of compromised private keys, we are convinced that many API secrets are also functional.

Causes & Mitigation Opportunities: We have seen both creators *actively* copying secrets from their local file system into the image, e.g., most of the API secrets but also private keys, incl. certificates, and *passively* generating key material during the image creation process, e.g., by installing an OpenSSH server. Both behaviors lead to compromised secrets and affect the security of both image creators and users basing their containers on an image and

already included secrets. Most likely, creators and users are unaware of compromising or using compromised foreign secrets. In fact, compared to GitHub, which provides a graphical interface to browse published files and potentially notice a mistakenly uploaded secret, files in Docker images and containers cannot be browsed easily, i.e., users barely get an overview on included files. Furthermore, while Git repositories only include manually added files, images of Docker containers contain a complete system directory tree. Thus, files with included secrets cannot be identified.

The mitigation of these problems must be two-fold. On the one hand, image creators must be warned that they are uploading their secrets to (publicly reachable) Docker registries. On the other hand, when deploying containers based on downloaded images, users should be informed that included secrets, especially private keys, might already be compromised, putting the authentication of deployed services at stake. To this end, credential-finding tools such as TruffleHog [76] or SecretScanner [14] can be integrated on both sides of the Docker paradigm. When uploading or downloading an image, these tools could then scan all layers of the image for included secrets. To reduce the number of false positives, for potential API secrets, the tool can also check the secret's function against the respective endpoint (we think this is also ethically correct on the user's side who downloaded the image). For private keys, the tools could maintain a list of test keys that are usually included in libraries. Increasing the image creator's awareness regarding the leakage of such secrets should decrease their number in uploaded images. Additionally, performing a second check at the user deploying a container based on a downloaded image should further decrease the number of services relying on already compromised secrets. An additional help could be an API + graphical view for images on Docker Hub, which shows the included files. This API could also enable third-party solutions similar to those for GitHub [31, 37, 76] to easily search for known secret file paths.

8 CONCLUSION

Containerization allows integrating applications and their dependencies in self-containing and shareable images making software deployment easy. However, when focusing on security, sharing of secrets or using already compromised secrets breaks promises, e.g., authenticity or access control. Thus, cryptographic secrets must not be included in publicly available container images.

Our analysis of 337,171 images from Docker Hub and 8,076 private registries revealed that, however, 8.5 % include secrets that should not be leaked to the public. More specifically, we found a near-lower bound of 52,107 private keys and 3,158 API secrets. 2,920 API secrets belonging to cloud providers, e.g., Amazon AWS API (1,213 secrets), or 25 secrets to financial services, e.g., Stripe API (18 secrets), show that attackers can cause immediate damage knowing these secrets. Focusing on the leaked private keys, we find that these are also in use in practice: 275,269 TLS and SSH hosts on the Internet rely their authentication on found keys, thus being susceptible to impersonation attacks. Notably, many private keys automatically generate when installing packages during image creation. While beneficial when running on real hardware where every computer generates its own key, in container images, this

process automatically leads to compromised secrets and potentially a sheer number of containers with compromised authenticity.

We further discover that especially private registries serve images with potentially sensitive software, most likely not intended to be publicly shared. Additionally, these registries might not prevent write access enabling attackers to add malware to images.

Our work shows that secret leakage in container images is a real threat and not neglectable. Especially the proven usage of leaked private keys in practice verifies numerous introduced attack vectors. As a countermeasure, the awareness of image creators and users regarding secret compromise must be increased, e.g., by integrating credential search tools into the Docker paradigm.

REFERENCES

[1] David Adrian, Karthikeyan Bhargavan, et al. 2015. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *ACM CCS*.
[2] Marco Balduzzi, Jonas Zaddach, et al. 2012. A security analysis of Amazon's Elastic Compute Cloud service. *IEEE/IFIP DSN* (2012).
[3] Giovanni Barbieri, Mauro Conti, et al. 2021. Assessing the Use of Insecure ICS Protocols via IXP Network Traffic Analysis. In *IEEE ICCCN*.
[4] Kelly Brady, Seung Moon, et al. 2020. Docker Container Security in Cloud Computing. In *IEEE CCWC*.
[5] Stuart Burns. 2021. How to keep Docker secrets secret. https://www.techtarget.com/searchitoperations/tip/How-to-keep-Docker-secrets-secret. (Accessed on 06/13/2022).
[6] Joao M. Ceron, Justyna J. Chromik, et al. 2020. Online Discoverability and Vulnerabilities of ICS/SCADA Devices in the Netherlands. arXiv:2011.02019.
[7] Taejoong Chung, Yabing Liu, et al. 2016. Measuring and Applying Invalid SSL Certificates: The Silent Majority. In *ACM IMC*.
[8] Theo Combe, Antony Martin, et al. 2016. To Docker or Not to Docker: A Security Perspective. *IEEE Cloud Comp.* 3, 5 (2016).
[9] COMSYS. 2023. Docker Secret Analysis Code. https://github.com/COMSYS/docker-secret-analysis.
[10] Ang Cui and Salvatore J. Stolfo. 2010. A Quantitative Analysis of the Insecurity of Embedded Network Devices: Results of a Wide-Area Scan. In *ACM ACSAC*.
[11] Markus Dahlmanns, Johannes Lohmöller, et al. 2020. Easing the Conscience with OPC UA: An Internet-Wide Study on Insecure Deployments. In *ACM IMC*.
[12] Markus Dahlmanns, Johannes Lohmöller, et al. 2022. Missed Opportunities: Measuring the Untapped TLS Support in the Industrial Internet of Things. In *ACM ASIACCS*. New York, NY, USA.
[13] Jean-Laurent de Morlhon. 2020. Scaling Docker's Business to Serve Millions More Developers: Storage - Docker. https://www.docker.com/blog/scaling-dockers-business-to-serve-millions-more-developers-storage/. (Accessed on 08/17/2022).
[14] deepfence. 2022. SecretScanner. https://github.com/deepfence/SecretScanner. (Accessed on 10/11/2022).
[15] David Dittrich and Erin Kenneally. 2012. *The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research*. Technical Report. U.S. Department of Homeland Security.
[16] Docker Inc. 2022. Docker Documentation: Best practices for writing Dockerfiles. https://docs.docker.com/develop/develop-images/dockerfile{_}best-practices/. (Accessed on 11/11/2022).
[17] Docker Inc. 2022. Docker Documentation: Deploy a registry server. https://docs.docker.com/registry/deploying/. (Accessed on 11/30/2022).
[18] Docker Inc. 2022. Docker Documentation: Dockerfile reference. https://docs.docker.com/engine/reference/builder/. (Accessed on 08/11/2022).
[19] Docker Inc. 2022. Docker Documentation: HTTP API. https://docs.docker.com/registry/spec/api/. (Accessed on 08/09/2022).
[20] Docker Inc. 2022. Docker Documentation: Image Manifest. https://docs.docker.com/registry/spec/manifest-v2-2/. (Accessed on 08/09/2022).
[21] Docker Inc. 2022. Docker Hub Container Image Library. https://hub.docker.com/. (Accessed on 06/07/2022).
[22] Docker Inc. 2022. Increase Rate Limits - Docker. https://www.docker.com/increase-rate-limits/. (Accessed on 08/17/2022).

[23] Docker Inc. 2022. Manage sensitive data with Docker secrets. https://docs.docker.com/engine/swarm/secrets/. (Accessed on 06/15/2022).

[24] Docker Inc. 2022. What is a Container? - Docker. https://www.docker.com/resources/what-container/. (Accessed on 08/09/2022).

[25] Zakir Durumeric, David Adrian, et al. 2015. A Search Engine Backed by Internet-Wide Scanning. In *ACM CCS*.

[26] Zakir Durumeric, Eric Wustrow, et al. 2013. ZMap: Fast Internet-wide Scanning and Its Security Applications. In *USENIX SEC*.

[27] Carlo Farinella, Ali Ahmed, et al. 2021. Git Leaks: Boosting Detection Effectiveness Through Endpoint Visibility. In *IEEE TrustCom*.

[28] Runhan Feng, Ziyang Yan, et al. 2022. Automated Detection of Password Leakage from Public GitHub Repositories. In *ACM ICSE*. New York, NY, USA.

[29] Oliver Gasser, Ralph Holz, et al. 2014. A deeper understanding of SSH: Results from Internet-wide scans. In *IEEE NOMS*.

[30] Béla Genge and Călin Enăchescu. 2016. ShoVAT: Shodan-Based Vulnerability Assessment Tool for Internet-Facing Services. *Sec. and Commun. Netw.* 9, 15 (2016).

[31] GitGuardian. 2022. Git Security Scanning & Secrets Detection. https://www.gitguardian.com/. (Accessed on 06/17/2022).

[32] Leonid Glanz, Patrick Müller, et al. 2020. Hidden in Plain Sight: Obfuscated Strings Threatening Your Privacy. In *ACM ASIACCS*. New York, NY, USA.

[33] Dan Goodin. 2013. PSA: Don't upload your important passwords to GitHub. https://arstechnica.com/information-technology/2013/01/psa-dont-upload-your-important-passwords-to-github/. (Accessed on 06/13/2022).

[34] Dan Goodin. 2018. Thousands of servers found leaking 750MB worth of passwords and keys. https://arstechnica.com/information-technology/2018/03/thousands-of-servers-found-leaking-750-mb-worth-of-passwords-and-keys/. (Accessed on 06/13/2022).

[35] Adam Hansson, Mohammad Khodari, et al. 2018. Analyzing Internet-connected industrial equipment. In *IEEE ICSigSys*.

[36] Nadia Heninger, Zakir Durumeric, et al. 2012. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *USENIX SEC*.

[37] Michael Henriksen. 2022. Reconnaissance tool for GitHub organizations. https://github.com/michenriksen/gitrob. (Accessed on 06/17/2022).

[38] Jens Hiller, Johanna Amann, et al. 2020. The Boon and Bane of Cross-Signing: Shedding Light on a Common Practice in Public Key Infrastructures. In *ACM CCS*.

[39] Ralph Holz, Johanna Amann, et al. 2016. TLS in the Wild: An Internet-wide Analysis of TLS-based Protocols for Electronic Communication. *NDSS* (2016).

[40] Ralph Holz, Lothar Braun, et al. 2011. The SSL Landscape: A Thorough Analysis of the x.509 PKI Using Active and Passive Measurements. In *ACM IMC*.

[41] Ralph Holz, Jens Hiller, et al. 2020. Tracking the Deployment of TLS 1.3 on the Web: A Story of Experimentation and Centralization. *ACM SIGCOMM Comput. Commun. Rev.* 50, 3 (2020).

[42] Delu Huang, Handong Cui, et al. 2019. Security Analysis and Threats Detection Techniques on Docker Container. In *IEEE ICCC*.

[43] Henri Hubert. 2021. Secrets exposed in Docker images: Hunting for secrets in Docker Hub. https://blog.gitguardian.com/hunting-for-secrets-in-docker-hub/. (Accessed on 06/13/2022).

[44] Vipin Jain, Baldev Singh, et al. 2021. Static Vulnerability Analysis of Docker Images. *IOP: Mat. Sc. and Eng.* 1131, 1 (apr 2021).

[45] Sabrina Kall and Slim Trabelsi. 2021. An Asynchronous Federated Learning Approach for a Security Source Code Scanner. In *ICISSP*, Paolo Mori, Gabriele Lenzini, and Steven Furnell (Eds.).

[46] Timo Kiravuo, Seppo Tiilikainen, et al. 2015. Peeking Under the Skirts of a Nation: Finding ICS Vulnerabilities in the Critical Digital Infrastructure. In *ECCWS*.

[47] Alexander Krause, Jan H. Klemmer, et al. 2022. Poster: Committed by Accident —-Prevention and Remediation Strategies Against Secret Leakage. https://www.ieee-security.org/TC/SP2022/program-posters.html.

[48] Deepak Kumar, Zhengping Wang, et al. 2018. Tracking Certificate Misissuance in the Wild. In *IEEE SP*.

[49] Mohit Kumar. 2013. Hundreds of SSH Private Keys exposed via GitHub Search. https://thehackernews.com/2013/01/hundreds-of-ssh-private-keys-exposed.html. (Accessed on 06/13/2022).

[50] Detectify Labs. 2016. Slack bot token leakage exposing business critical information. https://labs.detectify.com/2016/04/28/slack-bot-token-leakage-exposing-business-critical-information/. (Accessed on 06/15/2022).

[51] Hyunwoo Lee, Doowon Kim, et al. 2021. TLS 1.3 in Practice: How TLS 1.3 Contributes to the Internet. In *ACM WWW*. New York, NY, USA.

[52] Joonhee Lee, Hyunwoo Lee, et al. 2021. Analyzing Spatial Differences in the TLS Security of Delegated Web Services. In *ACM ASIACCS*. New York, NY, USA.

[53] Éireann P. Leverett. 2011. *Quantitatively Assessing and Visualising Industrial System Attack Surfaces.* Master's thesis. University of Cambridge.

[54] Guannan Liu, Xing Gao, et al. 2022. Exploring the Unchartered Space of Container Registry Typosquatting. In *USENIX SEC*.

[55] Peiyu Liu, Shouling Ji, et al. 2020. Understanding the Security Risks of Docker Hub. In *ESORICS*, Liqun Chen, Ninghui Li, Kaitai Liang, and Steve Schneider

(Eds.). Cham.

[56] S. Lounici, M. Rosa, et al. 2021. Optimizing Leak Detection in Open-Source Platforms with Machine Learning Techniques. In *ICISSP*.

[57] Federico Maggi, Rainer Vosseler, et al. 2018. *The Fragility of Industrial IoT's Data Backbone: Security and Privacy Issues in MQTT and CoAP Protocols.* Technical Report. Trend Micro Inc.

[58] Michael Meli, Matthew R. McNiece, et al. 2019. How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories. *NDSS* (2019).

[59] Ariana Mirian, Zane Ma, et al. 2016. An Internet-wide view of ICS devices. In *IEEE PST*.

[60] Marcin Nawrocki, Thomas C. Schmidt, et al. 2020. Uncovering Vulnerable Industrial Control Systems from the Internet Core. In *IEEE/IFIP NOMS*.

[61] Claus Pahl. 2015. Containerization and the PaaS Cloud. *IEEE Cloud Comp.* 2, 3 (2015).

[62] Akond Rahman, Chris Parnin, et al. 2019. The Seven Sins: Security Smells in Infrastructure as Code Scripts. In *ICSE*.

[63] Akond Rahman, Md Rayhanur Rahman, et al. 2021. Security Smells in Ansible and Chef Scripts: A Replication Study. *ACM Trans. Softw. Eng. Methodol.* 30, 1 (jan 2021).

[64] Akond Rahman and Laurie Williams. 2021. Different Kind of Smells: Security Smells in Infrastructure as Code Scripts. *IEEE S&P* 19, 3 (2021).

[65] Md Rayhanur Rahman, Akond Rahman, et al. 2019. Share, But be Aware: Security Smells in Python Gists. In *IEEE ICSME*.

[66] RedHunt Labs. 2021. Scanning Millions Of Publicly Exposed Docker Containers — Thousands Of Secrets Leaked (Wave 5). https://redhuntlabs.com/blog/scanning-millions-of-publicly-exposed-docker-containers-thousands-of-secrets-leaked.html. (Accessed on 06/13/2022).

[67] Aakanksha Saha, Tamara Denning, et al. 2020. Secrets in Source Code: Reducing False Positives using Machine Learning. In *IEEE COMSNETS*.

[68] Luca Schumann, Trinh Viet Doan, et al. 2022. Impact of Evolving Protocols and COVID-19 on Internet Traffic Shares. https://arxiv.org/abs/2201.00142.

[69] SecurityFail. 2022. kompromat. https://github.com/SecurityFail/kompromat. (Accessed on 11/09/2022).

[70] Matías Sequeira. 2020. Low-hanging Secrets in Docker Hub and a Tool to Catch Them All. https://ioactive.com/guest-blog-docker-hub-scanner-matias-sequeira/. (Accessed on 06/13/2022).

[71] Shodan. 2013. Shodan. https://www.shodan.io.

[72] Vibha Sinha, Diptikalyan Saha, et al. 2015. Detecting and Mitigating Secret-Key Leaks in Source Code Repositories. In *IEEE/ACM MSR*.

[73] Drew Springall, Zakir Durumeric, et al. 2016. Measuring the Security Harm of TLS Crypto Shortcuts. In *ACM IMC*.

[74] Stack Overflow. 2022. Developer Survey 2021. https://insights.stackoverflow.com/survey/2021. (Accessed on 07/11/2022).

[75] The Linux Foundation. 2022. Kubernetes - Production-Grade Container Orchestration. https://kubernetes.io/. (Accessed on 11/12/2022).

[76] TruffleSecurity. 2022. TruffleHog. https://github.com/trufflesecurity/trufflehog. (Accessed on 06/17/2022).

[77] Itamar Turner-Trauring. 21. Don't leak your Docker image's build secrets. https://pythonspeed.com/articles/docker-build-secrets/. (Accessed on 06/13/2022).

[78] Takahiro Ueda, Takayuki Sasaki, et al. 2022. An Internet-Wide View of Connected Cars: Discovery of Exposed Automotive Devices. In *ACM ARES*. New York, NY, USA.

[79] Abhishek Verma, Luis Pedrosa, et al. 2015. Large-Scale Cluster Management at Google with Borg. In *ACM EuroSys*. New York, NY, USA.

[80] Jinpeng Wei, Xiaolan Zhang, et al. 2009. Managing Security of Virtual Machine Images in a Cloud Environment. In *ACM CCSW*.

[81] Jonathan Codi West and Tyler Moore. 2022. Longitudinal Study of Internet-Facing OpenSSH Update Patterns. In *PAM*, Oliver Hohlfeld, Giovane Moura, and Cristel Pelsser (Eds.). Cham.

[82] Jordan Wrigtht. 2014. Why Deleting Sensitive Information from Github Doesn't Save You. https://jordan-wright.com/blog/2014/12/30/why-deleting-sensitive-information-from-github-doesnt-save-you/. (Accessed on 06/13/2022).

[83] Wei Xu, Yaodong Tao, et al. 2018. The Landscape of Industrial Control Systems (ICS) Devices on the Internet. In *IEEE Cyber SA*.

[84] Ahmed Zerouali, Tom Mens, et al. 2019. On the Relation between Outdated Docker Containers, Severity Vulnerabilities, and Bugs. In *IEEE SANER*.

[85] Ahmed Zerouali, Tom Mens, et al. 2021. On the usage of JavaScript, Python and Ruby packages in Docker Hub images. *Sc. of Comp. Prog.* 207 (2021).

[86] Nannan Zhao, Vasily Tarasov, et al. 2019. Large-Scale Analysis of the Docker Hub Dataset. In *IEEE CLUSTER*.

[87] Nannan Zhao, Vasily Tarasov, et al. 2019. Slimmer: Weight Loss Secrets for Docker Registries. In *IEEE CLOUD*.

[88] Zeljka Zorz. 2014. 10,000 GitHub users inadvertently reveal their AWS secret access keys. https://www.helpnetsecurity.com/2014/03/24/10000-github-users-inadvertently-reveal-their-aws-secret-access-keys/. (Accessed on 06/13/2022).

A ETHICAL CONSIDERATIONS

Our research curates a comprehensive archive of leaked security secrets in Docker images on Docker Hub and private registries whose leakage is again a threat to security. Moreover, to find private registries and deployments relying their security on leaked secrets, we leverage Internet-wide measurements that can have unintended implications, e.g., high load on single network connections impacting stability or alerting sysadmins due to unknown traffic. Thus, we base our research on several ethical considerations.

First, we take well-established guidelines [15] and best practices of our institution as base for our research. We handle all collected data with care and inform image creators and Docker Inc., to responsibly disclose our findings (cf. Appendix A.1). Moreover, we comply with recognized measurement guidelines [26] for our Internet-wide measurements reducing their impact (cf. Appendix A.2).

A.1 Handling of Data & Responsibilities

During our research, we always only collect and request publicly available data, i.e., our access is limited to publicly available image repositories. At no time do we bypass access control, e.g., by guessing passwords. We, thus, cannot download private images. Still, we revealed that many of the public images contain sensitive security secrets (cf. Section 5.1) which we stored for further analysis. All found secrets are stored on secured systems. Furthermore, we refrain from releasing our dataset including these secrets or image names, to not provide an archive of leaked secrets or pinpoints for potential attackers. While this restriction prevents others from independently reproducing our results, we consider this decision to constitute a reasonable trade-off to protect affected users.

Responsible Disclosure: To further support affected users in removing their secrets from publicly available Docker images, we target to responsibly disclose our findings. To this end, we extract e-mail addresses from maintainer variables set in Dockerfiles and furthermore derive addresses from Gravatar accounts linked to affected Docker Hub accounts. In this regard, we identified 1,181 e-mail addresses we contacted to notify about our possible findings. Already after a few hours, we received >30 answers of owners appreciating our efforts, fixing their images or informing us that the image at hand is not used anymore. A handful informed us that no secrets were leaked helping us to refine our filtering. Moreover, we decided to reach out to the operator of Docker Hub, i.e., Docker Inc., to discuss potential further disclosure to unidentifiable creators.

A.2 Reducing Impact of Measurements

To reduce the impact of our active Internet scans, we follow widely accepted Internet measurement guidelines [26].

Coordination: We coordinate our measurements with our Network Operation Center to reduce the impact on the Internet and to react correspondingly. Abuse emails are handled informing about the intent of our measurements and how to opt-out of our measurements. As part of this opt-out process, we maintain a blocklist to exclude IPs from our measurements.

External Information: For giving external operators information about our research intent, we provide rDNS records for all our scan IPs and transmit contact information in the HTTP header of each request to the registries. Moreover, we host a webpage on our scan IPs, which gives further information on our project and how to opt-out. Over time, also due to other measurements, we excluded 5.8 M IP addresses (0.14% of the IPv4 address space).

Limiting Load: To limit load and stress on all systems involved (along the path and the end-host), we deliberately reduce our scan-rate. Our scans are stretched over the course of one day and use zmap's address randomization to spread load evenly. We further limit the load on single private registries when downloading available images. While we paid to increase the existing rate limiting for image downloads on Docker Hub (cf. Appendix B), private registries typically do not implement any rate limiting. Hence, to prevent our scanner from overloading registries running on resource-constrained hardware or connected via slow or volume-billed Internet connections, we decide to only download image layers randomly until their size sums up to at most 250 MB. Additionally, we shuffle the downloads of layers of different registries to further distribute the load.

A.3 Overall Considerations

Without taking our goals into account, summarizing the sensitive nature and the impact of our measurements can quickly lead to the conclusion that our measurements are not beneficial. However, we consider it public interest and fundamental for improving security to know about potential security issues and how widespread these are. The Docker paradigm does not include any mechanisms to prevent image creators from (accidentally) adding security secrets to their images and no mechanisms exist that warns users relying on already compromised security secrets. Hence, we consider it essential to know whether secrets are widely included in publicly available Docker images and whether these are in use at scale to steer future decisions for counter-measures. To answer this question, we carefully weighed the impact of our measurements against their benefit and have taken sensible measures to reduce the risks of building a large archive of leaked security secrets and risks introduced by active Internet measurements.

B IMAGE DOWNLOAD FROM DOCKER HUB

The limit of image manifest downloads from Docker Hub depends on the booked plan, e.g., free users are allowed to pull only 800 images per day. Hence, for a faster analysis of images on Docker Hub, we purchased two Pro accounts, that allow 5,000 image downloads per day each. Still, we are required to perform our analysis on a subpart of available images as the download of one image of every of the 9,321,726 available repositories would require 933 days under best conditions. Thus, we decided to limit our analysis on two categories: (i) a context of standard protocol and frequently used technologies, and (ii) an (Industrial) IoT context for comparison. Both categories have communication in common as here security can be affected on an Internet scale.

Standard Context: To generate a wide view on secret leakage in Docker images, we create a list of search queries comprising standard protocols [68], and frequently used technologies [74]. To find related images, we employ Docker Hub's API to perform searches over all available images and retrieve results users would retrieve when using the docker search CLI command or Docker Hub's web interface. To ensure that different handling of special characters in

Table 4: Search queries and derived spellings to receive corresponding Docker repositories from Docker Hub of our Standard and (Industrial) IoT query group.

Standard: *Trending protocols and technologies.*

· tls	· ipp	· css	· imap	· html	· mysql	· oracle	· mariadb	· memcached	· elasticsearch	
· ssh	· vpn	· sql	· pptp	· java	· mssql	· heroku	· ansible	· terraform	· c++ → c+, c, c	
· dns	· irc	· php	· xmpp	· bash	· redis	· docker	· xamarin	· postgresql	· ibm db2 → ibmdb2, ibm+db2	
· ftp	· aws	· quic	· yarn	· ipmi	· shell	· puppet	· firebase	· kubernetes	· unity 3d → unity+3d, unity3d	
· rdp	· gcp	· http	· deno	· samba	· proxy	· pulumi	· dynamodb	· javascript	· ibm cloud → ibmcloud, ibm+cloud	
· vnc	· git	· smtp	· chef	· rsync	· telnet	· python	· cassandra	· typescript	· node.js → node js, node+js, nodejs	
· smb	· k8s	· pop3	· flow	· ipsec	· sqlite	· mongodb	· couchbase	· powershell	· ibm watson → ibm+watson, ibmwatson	
· ipp	· css	· imap	· html	· mysql						

(Industrial) IoT: *Industrial protocols subject to recent research.*

· atg	· mqtt	· codesys	· ff-hse → ff hse, ff+hse, ffhse	· iec-61850 → iec+61850, iec61850, iec 61850	
· dnp3	· cspv4	· proconos	· fl-net → fl net, flnet, fl+net	· zigbee-ip → zigbeeip, zigbee ip, zigbee+ip	
· srtp	· bacnet	· ethercat	· hart-ip → hart+ip, hartip, hart ip	· ansi c12.22 → ansi c12 22, ansi+c12+22, ansic1222	
· iccp	· modbus	· profinet	· iec-104 → iec 104, iec104, iec+104	· ethernet/ip → ethernet+ip, ethernetip, ethernet ip	
· amqp	· siemens	· pc worx → pcworx, pc+worx	· omron fins → omron+fins, omronfins	· red lion crimson v3 → redlioncrimsonv3, red+lion+crimson+v3	
· coap	· tridium	· opc-ua → opcua, opc+ua, opc ua	· melsec-q → melsecq, melsec q, melsec+q	· automatic tank gauge → automatic+tank+gauge, automatictankgauge	

technology and protocol names does not exclude any images, we include different spelling variants in our query list, i.e., we include terms as they are, but also replace non-alpha-numeric characters by + and *space*. Table 4 (top) shows our constructed search queries for the standard context.

(Industrial) IoT Context: We extend our analysis on images in the (Industrial) IoT context, as deployments in this area showed massive security deficits in past [11, 12, 35, 53, 59, 60], in single cases traced back to security secret leakage via GitHub and Docker images [12]. As search terms, we take (Industrial) IoT protocol names that were subject to recent research [12]. We proceed similar as in the standard context, i.e., include derived spellings of these terms, and show our constructed search query of this context in Table 4 (bottom).

C REGULAR EXPRESSIONS

Following already established procedures to find security secrets in code repositories [58, 76], we build our secret detection in Docker Images on regular expressions, i.e., we try to match regular expressions derived from secrets on the content of included files. Table 5 shows our composed list of regular expressions covering a variety of secrets, i.e., asymmetric private keys and API keys, as well as accompanying material we use for our analysis, i.e., public keys and certificates. We orientate our expressions towards related work [58] and TruffleHog [76], an established tool to find secrets in various sources, i.e., the local file system, Git repositories, S3 storages, and syslogs. Specifically, we inherit Meli et al.'s [58] regular expressions to allow comparisons between the occurrence of leaked secrets in GitHub repositories at scale and our findings. Furthermore, they composed their expressions comprehensibly, i.e., they included API keys for certain services by the occurrence of service domains in Alexa's Top 50 Global and United States lists in combination with a list of well-known APIs manually filtered for services with a high risk on key leakage and keys with a distinctive signature (to reduce the number of false-positives). For private keys they focus on the most prevalent types and form to store, i.e., RSA, elliptic curve keys, PGP, and general keys in PEM format.

To spread our analysis and align our expressions to the scope of our search queries (cf. Appendix B), we adapt our expression for private keys to match every type of private key in PEM format and, furthermore, extend the list of expressions to also match private key

blocks, keys in PKCS7 format, and keys stored in XML format (due to their unambiguous signature). Regarding API secrets to match, we extend our list with expressions from TruffleHog [76] on basis of services being currently trending under developers [74] or having a high risk for misuse and the regular expressions including a unique signature (also to reduce the number of false positives). For some services we found more than one type of secret, i.e., secrets for different API versions (GitHub v1 and v2), or different types of keys (Stripe). Our final list contains 48 expressions which we match on the content of every file in the images part of our study.

D FILTERING BASED ON FILEPATHS

After matching our regular expressions on arbitrary file content available in Docker images, extensive filtering is required to exclude false positive matches, i.e., matches that do not contain any secret. Our *File* filter bases on file paths derived from matches our *Kompromat* filter excluded, i.e., all parent directories under which we find more than $2/3$ test keys known by kompromat [69] and all directories that include known test keys directly. Additionally, it takes manually compiled file paths, e.g., where standard libraries reside (`/var/lib/*`) or package managers store their downloads (e.g., `*/.cache/pip/*`) and extensions of database files (e.g., `db` and `dbf`) into account which we selected after manually revisit all matches as these produced a high number of false positives.

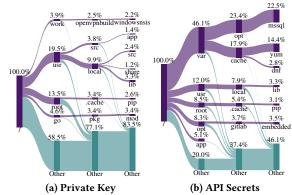**(a) Private Key**   **(b) API Secrets**

Figure 7: Most frequent file paths suspect to our filtering based on file path or extension.

Table 5: Regular expressions we matched on each file's content in the layers and environment variables of selected images.

Domain	Name	Subordinate	Expression	Source
Private Key	PEM Private Key		(?i)−−\s*?BEGIN[A-Z0-9_-]*?PRIVATE KEY\s*?−−[-a-zA-Z0-9\/\n\r=+]* −−\s*?END[A-Z0-9_-]*? PRIVATE KEY\s*?−−	own
	PEM Private Key Block		(?i)−−\s*?BEGIN[A-Z0-9_-]*?PRIVATE KEY BLOCK\s*?−−[-a-zA-Z0-9\/\n\r=+]* −−\s*?END[A-Z0-9_-]*? PRIVATE KEY BLOCK\s*?−−	
	PEM PKCS7		(?i)−−\s*?BEGIN PKCS7\s*?−−[-a-zA-Z0-9\/\n\r=+]*−−\s*?END PKCS7\s*?−−	
	XML Private Key		(?i)<(RSAKeyValue|DSAKeyValue|ECKeyValue)>(.|[\n\r])+<\/RSAKeyValue|\/DSAKeyValue|\/ECKeyValue)>	
API · Cloud	Alibaba		\b(LTAI[a-zA-Z0-9]{17,21})[\"'.;\s]*	[76]
	Amazon AWS		\b((?:AKIA|ABIA|ACCA|ASIA)[0-9A-Z]{16})\b	[76]
	Azure		(?i)(client_secret|clientsecret).{0,20}([a-z0-9_\.\-~]{34})	[76]
	DigitalOcean		(?i)(?:digitalocean)(?:.|[\n\r]){0,40}\b([A-Za-z0-9_-]{64})\b	[76]
	Github		\b((?:ghp|gho|ghu|ghs|ghr)_[a-zA-Z0-9]{36,255})\b	[76]
	Gitlab	v1	(?i)(?:gitlab)(?:.|[\n\r]){0,40}\b([a-zA-Z0-9\-=_]{20,22})\b	[76]
		v2	\b(glpat-[a-zA-Z0-9\-=_]{20,22})\b	
	Google Cloud		\{[^{}]+auth_provider_x509_cert_url[^}]+\}	[58]
	Google Services		\bAIza[0-9A-Za-z\-_]{35}\b	[58]
	Heroku		(?i)(?:heroku)(?:.|[\n\r]){0,40}\b([0-9Aa-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12})\b	[76]
	IBM Cloud Identity Services		(?i)(?:ibm)(?:.|[\n\r]){0,40}\b([A-Za-z0-9_-]{44})\b	[76]
	Login Radius		(?i)(?:loginradius)(?:.|[\n\r]){0,40}\b([0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12})\b	[76]
	MailChimp		\b[0-9a-f]{32}-us[0-9]{1,2}\b	[58]
	MailGun		\bkey-[0-9a-zA-Z]{32}\b	[58]
	Microsoft Teams		(https:\/\/[a-zA-Z-0-9]+\.webhook\.office\.com\/webhookb2\/[a-zA-Z-0-9]{8}-[a-zA-Z-0-9]{4}- [a-zA-Z-0-9]{4}-[a-zA-Z-0-9]{4}-[a-zA-Z-0-9]{12}\/@[a-zA-Z-0-9]{8}-[a-zA-Z-0-9]{4}- [a-zA-Z-0-9]{4}-[a-zA-Z-0-9]{4}-[a-zA-Z-0-9]{12}\/IncomingWebhook\/[a-zA-Z-0-9]{32}\/[a-zA-Z-0-9]{8}-[a-zA-Z-0-9]{4}- [a-zA-Z-0-9]{4}-[a-zA-Z-0-9]{4}-[a-zA-Z-0-9]{12})	[76]
	Netlify		(?i)(?:netlify)(?:.|[\n\r]){0,40}\b([A-Za-z0-9_-]{43,45})\b	[76]
	Twilio		\bSK[0-9a-fA-F]{32}\b	[58]
API · Financial	Amazon MWS		\bamzn\.mws\.[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}\b	[58]
	Bitfinex		(?i)(?:bitfinex)(?:.|[\n\r]){0,40}\b([A-Za-z0-9_-]{43})\b	[76]
	Coinbase		(?i)(?:coinbase)(?:.|[\n\r]){0,40}\b([a-zA-Z-0-9]{64})\b	[76]
	Currency Cloud		(?i)(?:currencycloud)(?:.|[\n\r]){0,40}\b([a-zA-Z0-9\-=_]{20,22})\b	[76]
	Paydirt		(?i)(?:paydirtapp)(?:.|[\n\r]){0,40}\b([a-z0-9]{32})\b	[76]
	Paymo		(?i)(?:paymoapp)(?:.|[\n\r]){0,40}\b([a-zA-Z0-9]{44})\b	[76]
	Paymongo		(?i)(?:paymongo)(?:.|[\n\r]){0,40}\b([a-zA-Z0-9_]{32})\b	[76]
	PayPal Braintree		\baccess_token\$production\$[0-9a-z]{16}\$[0-9a-f]{32}\b	[58]
	Picatic		\bsk_live_[0-9a-z]{32}\b	[58]
	Stripe	ST	\bsk_live_[0-9a-zA-Z]{24}\b	[58]
		RE	\brk_live_[0-9a-zA-Z]{24}\b	
	Square	AT	\bsq0atp-[0-9A-Za-z\-_]{22}\b	[58]
		OA	\bsq0csp-[0-9A-Za-z\-_]{43}\b	
	Ticketmaster		(?i)(?:ticketmaster)(?:.|[\n\r]){0,40}\b([a-zA-Z-0-9]{32})\b	[76]
	WePay		(?i)(?:wepay)(?:.|[\n\r]){0,40}\b([a-zA-Z-0-9_?]{62})\b	[76]
API · Social Media	Facebook	Key	\b([A-Za-z0-9_\.]{69}-[A-Za-z0-9_\.]{10})\b	[76]
		Key	\bEAACEdEose0cBA[0-9A-Za-z]+\b	[58]
	Twitter		\b[1-9][0-9]+-[0-9a-zA-Z]{40}\b	[58]
API · IoT	Accuweather		(?i)(?:accuweather)(?:.|[\n\r]){0,40}([a-z0-9A-Z\%]{35})\b	[76]
	Adafruit IO		\b(aio_[a-zA-Z0-9]{28})\b	[76]
	OpenUV		(?i)(?:openuv)(?:.|[\n\r]){0,40}\b([0-9a-z]{32})\b	[76]
	Tomtom		(?i)(?:tomtom)(?:.|[\n\r]){0,40}\b([0-9Aa-zA-Z]{32})\b	[76]
Accompanying Material	PEM Certificate		(?i)−−\s*?BEGIN CERTIFICATE\s*?−−[-a-zA-Z0-9\/\n\r=+]*−−\s*?END CERTIFICATE\s*?−−	own
	PEM Certificate Request		(?i)−−\s*?BEGIN CERTIFICATE REQUEST\s*?−−[-a-zA-Z0-9\/\n\r=+]* −−\s*?END CERTIFICATE REQUEST\s*?−−	
	PEM Public Key		(?i)−−\s*?BEGIN[A-Z0-9_-]*?PUBLIC KEY\s*?−−[-a-zA-Z0-9\/\n\r=+]* −−\s*?END[A-Z0-9_-]*? PUBLIC KEY\s*?−−	
	PEM Public Key Block		(?i)−−\s*?BEGIN[A-Z0-9_-]*?PUBLIC KEY BLOCK\s*?−−[-a-zA-Z0-9\/\n\r=+]* −−\s*?END[A-Z0-9_-]*? PUBLIC KEY BLOCK\s*?−−	
	SSH Host Key		\bssh-[0-9a-zA-Z]+ AAAA\S+ \S+\b	

Figure 7 shows the seven most prevalent file paths that contain matches excluded by our *File* filter. Indeed, most of the exclusions are matches included in folders belonging to package managers and thus most likely test secrets. The massive filtering of API secret matches in /var/opt/mssql is due to the high number of false positives of the Twitter regular expressions on database files.