DEIM Forum 2021 F33-4

Dockerfile 自動更新システムのためのバージョンごとの利用率および脆弱性情報を用いたベースイメージ推薦手法の検討

北島　信哉†　　関口　敦二†

† 株式会社富士通研究所 ソフトウェア研究所　〒 211–8588 川崎市中原区上小田中 4-1-1
E-mail: †{kitajima.shinya,sekia}@fujitsu.com

あらまし　近年，Docker コンテナを利用したアプリケーションのデプロイ手法が注目を集めている．Docker コンテナは起動が高速かつ動作が軽量であり，アプリケーションのポータビリティや再現性を向上できるため，CI/CD や DevOps といったリリースサイクルを高速化する手法とともによく用いられている．一方で，一度作成された Docker イメージが更新されなければ，セキュリティ上のリスクになったり，最新の機能が利用できないなどの問題が発生する．Docker イメージを作成する際は，通常，公開されている Docker イメージをベースイメージとして利用し，ベースイメージに対して必要な設定やファイルの追加などを行い，利用する Docker イメージを作成する．ベースイメージが公開されるたびに機械的に最新バージョンへの更新を推薦する手法も考えられるが，この手法ではユーザが更新の必要性を判別しづらいという課題があった．本稿では，Docker イメージの脆弱性情報や，GitHub で公開されている Dockerfile を利用したバージョンごとのユーザの比率の推定結果を用いて，ユーザに対してベースイメージの更新を促す手法について述べる．

キーワード　Docker コンテナ，Dockerfile，DevOps，自動化，バージョンアップ

1 はじめに

近年，Docker コンテナ [1]を利用したアプリケーションのデプロイ手法が注目を集めている．Docker コンテナは，Dockerfile をもとにビルドされた Docker イメージを利用し，コンテナ型仮想化技術によってアプリケーションを動作させる仕組みである．サーバ仮想化技術と異なり，アプリケーションの起動までに要する時間が非常に短いことや，同じ Docker イメージを他のサーバなどに移動して Docker コンテナを起動しても同じように動作することから，CI/CD や DevOps といったリリースサイクルを高速化する手法との相性がよく，アプリケーション開発時に採用される事例が非常に増加している [3], [6]．コンテナ型仮想化技術は Docker の登場以前から存在していたが，Docker イメージによりアプリケーションのポータビリティが向上したことや，Docker Hub[2]のような Docker イメージ共有システムができたことにより，一気に普及した [1], [7]．

一方で，Docker コンテナを利用するにあたり，いくつかの課題がある．課題の 1 つに，Docker イメージの更新が挙げられる [1]．Docker イメージのもととなる Dockerfile では，Docker イメージの生成に必要な手順を独自の書式で記述する必要があるが，その中には Docker イメージ作成時に親イメージとして利用するベースイメージや，アプリケーション動作に必要なライブラリ，利用するアプリケーションなど，バージョン管理されているものが含まれていることが多い．これらはできる限り定期的に最新のバージョンに更新することで，セキュリティリスクを軽減できるとともに，大幅なバージョンアップを避けることでバージョンアップが失敗した際の原因追求を容易できるといったメリットがあると考えられる．しかし，多数の Docker コンテナを生成・管理している場合，Docker イメージごとにベースイメージや依存ライブラリ，利用アプリケーションなどの更新を確認し，更新されていた場合に問題がなければ Dockerfile を更新してそのバージョンに書き換える，といった作業を人手で定期的に行うことは非常に煩雑である．

筆者らはこれまでに，ベースイメージを機械的に最新版に更新するため，ベースイメージに更新があった際に Dockerfile を自動で書き換えるとともに，ユーザに通知する手法の提案と評価をおこなっている．この手法では，ユーザは通知を受け取ると，更新された Dockerfile に問題がないか確認し，問題がなければ更新された Dockerfile をもとに新しい Docker イメージを生成する [4]．しかし，この手法では，最新版のベースイメージに脆弱性や不具合などが含まれる場合にユーザが気づくことが難しかったり，通知を受け取ったタイミングでベースイメージを更新すべきかどうかの判断が難しいという課題があった．

そこで本稿では，Docker Hub の情報や，GitHub[3]上で公開されている Dockerfile をもとに，利用しているユーザ数が多いと推定されるバージョンのベースイメージに更新する手法や，Docker イメージの脆弱性チェックの結果をもとにセキュリティ上問題のあるベースイメージを更新する手法について考察する．

以下，2 で想定環境について述べ，3 で提案手法について説明する．4 で提案手法について実験を行い，最後に 5 で本稿のまとめと今後の課題について述べる．

1：https://www.docker.com/
2：https://hub.docker.com/
3：https://github.com/

```
FROM python:3.5 # ベースイメージの特定（python:3.5）

# ベースイメージに追加する手順を以下に列挙
WORKDIR /usr/src/app
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt
COPY . .

CMD [ "python", "-u", "./main.py" ]
```

図 1　Dockerfile の例

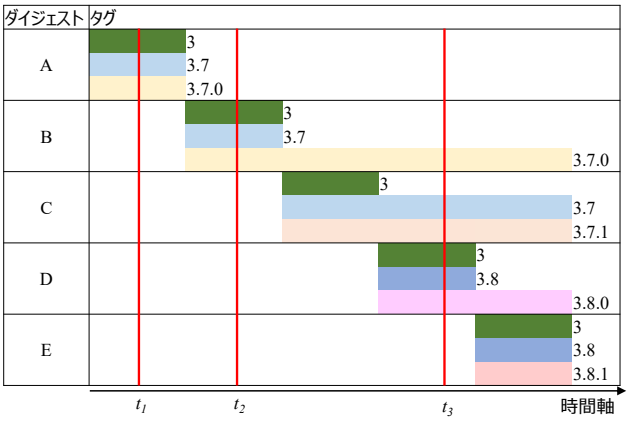

図 2　タグとダイジェストの関係

2　想 定 環 境

2.1　Dockerfile と Docker レジストリ

Docker イメージは，Dockerfile をビルドすることで生成できる．図 1 に Dockerfile の例を示す．Dockerfile では，初めに “FROM” で始まる行でベースイメージを指定する．図では “python:3.5” というベースイメージを指定している．ベースイメージは，基本的には Docker Hub と呼ばれるインターネットで公開されている Docker レジストリ（パブリックレジストリ）上で公開されている Docker イメージから選択して Dockerfile に記述して利用する．Docker イメージ生成の際に親イメージとして指定する Docker イメージをベースイメージと呼ぶが，ベースイメージは特別な Docker イメージではなく，どのような Docker イメージであってもベースイメージとして利用でき，ベースイメージから新たに作成した Docker イメージもまたベースイメージとして利用できる．作成した Docker イメージは，Docker レジストリにプッシュ（アップロード）しておくことで，他のユーザやマシン上からプル（ダウンロード）して利用できる．

パブリックレジストリとしては，Docker 社が運営する Docker Hub が最も有名であるが，それ以外にも，Red Hat 社が運営する Quay.io[4] や Google 社が運営する Google Container Registry[5] などが存在する．これらのパブリックレジストリの運営企業や，他のクラウドサービスの運営企業は，サービスとしてプライベートな Docker レジストリ（プライベートレジストリ）も提供している．パブリックレジストリ上の Docker イメージは誰でも利用できるが，プライベートレジストリは主に組織内でのみ利用する Docker イメージを置いておくためのもので，限られたユーザのみがアクセスできる．プライベートレジストリは，Docker Registry[6]，Harbor[7] などのソフトウェアを利用して任意の場所で動作させることもできる．

Dockerfile では，Docker Hub 上の Docker イメージ以外にも，これらのパブリックレジストリ，プライベートレジストリ上の Docker イメージをベースイメージとして指定して利用できる．本稿における提案手法では，大多数の Docker イメージが公開されている Docker Hub 上の Docker イメージをベースイメージとして利用することを想定している．他のパブリックレジストリやプライベートレジストリ上の Docker イメージをベースイメージとして利用する場合，脆弱性のチェックは可能であるが，ユーザ数が非常に少ない Docker イメージの場合，利用しているユーザ数の多いバージョンを推定することは難しい．

2.2　タグとダイジェスト

Docker イメージを指定する際に “python:3.5” と指定すると，Docker イメージ名が “python” である Docker イメージのうち，“3.5” というタグが付与されている Docker イメージが利用される．また，Docker イメージには固有のハッシュ値（ダイジェスト）が付与される．まったく同一の Docker イメージであればダイジェストは等しいが，そうでない場合は異なるダイジェストとなる．各 Docker イメージには任意のタグを付与することができる．図 2 に，タグとダイジェストの関係を示す．1 つの Docker イメージに対して複数のタグを付与できるが，1 つのタグに紐付けられる Docker イメージは 1 つのみである．また，タグは別の Docker イメージに付与し直すことができる．このとき，以前そのタグを付与されていた Docker イメージからはタグが解除される．

Docker イメージに付与されるタグは，その Docker イメージに含まれるアプリケーションのバージョンを表していることが多い．例えば，“python:3.5.0” という Docker イメージは，python のバージョン 3.5.0 が含まれる Docker イメージであることを表している．アプリケーションに対するバージョン番号の付与方法としては，セマンティックバージョニング [8] と呼ばれる手法が広く用いられている [2], [5]．セマンティックバージョニングでは，$X.Y.Z$ の形式でバージョン番号を付与し，X はメジャーバージョン，Y はマイナーバージョン，Z はパッチバージョンと呼ばれる．メジャーバージョン X の変更は後方互換性のない変更，マイナーバージョン Y の変更は後方互換性を保った変更，パッチバージョン Z の変更は後方互換性を保ったバグ修正を表す．提案手法では，主にセマンティックバージョ

4 : https://quay.io/
5 : https://cloud.google.com/container-registry/
6 : https://github.com/docker/distribution
7 : https://github.com/goharbor/harbor

8 : https://semver.org/

```
FROM [--platform=<platform>] <image> [AS <name>]
FROM [--platform=<platform>] <image>[:<tag>] [AS <name>]
FROM [--platform=<platform>] <image>[@<digest>] [AS <name>]
```

図 3　Dockerfile における FROM の記法

ニングを採用してタグが付与されている Docker イメージを想定する．

図 2 は，t_1 の時点ではダイジェスト A の Docker イメージに対して，同時に “3”，“3.7”，“3.7.0” の 3 つのタグが付与されていることを表している．次に，それぞれのタグはダイジェスト B の Docker イメージに付与し直されている．t_2 の時点で “3” や “3.7”，“3.7.0” というタグを指定してこの Docker イメージをダウンロードすると，ダイジェスト B の Docker イメージがダウンロードされる．また，t_3 の時点で “3” というタグを指定してこの Docker イメージをダウンロードすると，ダイジェスト D の Docker イメージがダウンロードされる．つまり，同じ “3” というタグを指定していても，ダウンロードするタイミングによって python のバージョンは異なる．

Dockerfile を記述する際のベースイメージの記法は，3 通り存在する [9]．図 3 に 3 通りのベースイメージの記法を示す．いずれも “FROM” から始まり，Docker イメージ名を記述するところまでは同じだが，タグやダイジェストを省略するか，タグを書くか，またはダイジェストを書くかで 3 通りにわかれている．また，Dockerfile では “#” から改行するまでの部分はコメントと見なされ，無視される．

Dockerfile をもとに Docker イメージをビルドする際，ベースイメージに指定した Docker イメージがビルドする環境内に存在しない場合，初めにベースイメージが Docker レジストリからダウンロードされる．一方で，ベースイメージに指定した Docker イメージがビルドする環境内に存在する場合，Docker レジストリから Docker イメージはダウンロードされず，環境内の Docker イメージがキャッシュとして利用される．

ここで，ベースイメージが環境内に存在するかどうかは，Dockerfile 内で指定した Docker イメージ名とタグの両方が一致する Docker イメージが環境内に存在するかどうかをチェックしており，ダイジェストが一致するかどうかまではチェックされない．ただし，ベースイメージを指定する際にダイジェストを指定している場合は，ダイジェストが一致するかどうかがチェックされる．また，Dockerfile 内で Docker イメージ名のみを指定してタグやダイジェストを指定しない場合，自動的に latest タグが付いているイメージを指定したことになる．このように，Docker イメージを作成するときに利用されるベースイメージは，Dockerfile をビルドしたときに決定される．GitHub 上で公開されている Dockerfile を分析した結果，Dockerfile 内でベースイメージを指定する際は Docker イメージ名とタグを指定することがケースが大半であることがわかっている．

9 : https://docs.docker.com/engine/reference/builder/

ベースイメージがキャッシュされてしまうと，キャッシュがない環境でのみ最新版のベースイメージがダウンロードされてしまうということが起きうる．CI/CD や DevOps においては，ビルドした環境によらず同一の成果物を得られることが望ましいため，本稿ではキャッシュを利用せずにビルドするたびに最新のベースイメージを Docker レジストリからダウンロードしてビルドするものとする．また，ユーザは α 版や β 版，リリース候補版のような開発版のタグが付与された Docker イメージは利用しないと想定する．このような開発版の Docker イメージは主に動作確認用に公開されており，通常のバージョンアップでは利用しないと考えられるためである．

Docker イメージのタグには，“3.8.2-buster” や “3.8.2-alpine3.11” のように OS などの付加情報が含まれるタグも存在する．本稿では，“セマンティックバージョン” を表している部分を “バージョン”，“OS などの付加情報” を表している部分を “タイプ” と呼ぶ．

2.3　Dockerfile の管理方法

Dockerfile は Docker イメージのビルド手順をコードで書き下したものであるため，Git[10] などのバージョン管理システムとの相性がよい．特に，アプリケーションの開発やオープンソースソフトウェア（OSS）の開発において，成果物を Docker イメージの形で利用・配布する場合，ソースコードと同じ Git リポジトリに Dockerfile を格納することが多い．また，パブリックレジストリで公開されている Docker イメージをもとに処理を加えて新たな Docker イメージを生成する場合も，作成した Dockerfile を Git リポジトリに格納することが多い．これは，バージョン管理システムを利用して Dockerfile の版数管理をするためだけでなく，CI ツールと連携してソースコードや Dockerfile を更新した際に自動で Docker イメージをビルドするという目的もあると考えられる．

本稿では，ユーザは CI/CD を行っており，Dockerfile は GitHub や GitLab[11] などのバージョン管理システム（Git リポジトリ）上で管理されていると想定している．GitHub はオンラインサービスのみしか存在しないが，GitLab はオンラインサービス以外にも OSS として公開されているバージョンが存在し，ユーザがローカル環境などで独自にサービスを立ち上げることが可能である．また，Git リポジトリは，誰でもアクセス可能なパブリックリポジトリと，限られたユーザのみがアクセスできるプライベートリポジトリの 2 種類が存在する．提案手法は，ユーザが Dockerfile を配置するリポジトリがパブリックリポジトリであってもプライベートリポジトリであっても適用可能である．一方で，ユーザが Dockerfile を Git リポジトリ上で管理していない場合であっても，提案手法を実行するシステムから Dockerfile に対してアクセスが可能であれば適用できると考えている．

10 : https://git-scm.com/
11 : https://about.gitlab.com/

2.4 提案手法の提供形態

提案手法の提供形態としては，以下の 2 つを想定している．

（1） SaaS（Software as a Service）のように外部サーバ上でサービスとして提供し，各ユーザの Dockerfile が配置されている Git リポジトリにアクセスする形態

（2） ユーザ側のサーバ上で動作するアプリケーションとして配布し，各ユーザの Dockerfile が配置されている Git リポジトリにアクセスする形態

(1) の提供形態は，Dockerfile がインターネットから直接アクセス可能な Git リポジトリに配置されている場合に適している．一方で，(2) の提供形態は，Dockerfile がインターネットから直接アクセスできない企業内や組織内の Git リポジトリに配置されている場合に適しているが，Dockerfile がインターネットから直接アクセス可能な Git リポジトリに配置されている場合であっても利用が可能であるというメリットがある．しかし，ユーザ側のサーバ上で提案手法を実施するためのアプリケーションを動作させる必要があるため，安定稼働させるための冗長化が困難であったり，アプリケーションを稼働させるためのサーバを用意する必要があるといったデメリットが存在する．

3 提 案 手 法

本章では，GitHub 上で公開されている Dockerfile を収集して分析することで，Docker イメージ利用者がどのタグの Docker イメージを最も利用しているかを調べ，その結果をもとにユーザに対してベースイメージの更新を推薦する手法と，Docker イメージをダウンロードして脆弱性チェックをおこない，脆弱性が発見された場合には他のタグの Docker イメージに更新する手法について述べる．

3.1 ユーザ比率推定手法

本節では，Docker イメージ利用者がどのタグの Docker イメージを利用しているかを推定した結果をもとに，タグの更新を推薦する手法について述べる．

3.1.1 GitHub 上で公開されている Dockerfile の収集

GitHub では GitHub REST API v3[12]が公開されており，その中にコード検索用の API が含まれている．この検索 API ではソースコードの言語を指定してコードを検索でき，“Dockerfile” を指定して検索することで，Dockerfile のみを検索できる．例えば python という Docker イメージをベースイメージに利用している Dockerfile を収集したい場合，言語に “Dockerfile” を指定し，“FROM python” という検索語で検索することで，目的の Dockerfile を検索できる．

しかし，GitHub 上で公開されている Dockerfile を収集したい場合，検索 API では以下のような問題がある．

- 検索 API は結果を最大で 100 件ずつ 10 ページまで，すなわち最大で上位 1000 件までしか返さないため，それ以上の検索結果が存在する場合，すべての検索結果を収集できない．

12 : https://docs.github.com/en/free-pro-team@latest/rest

表 1 ベースイメージに node を利用した Dockerfile の日付ごとの収集数（横方向：収集日，縦方向：コミット日，*：分析に利用する Dockerfile）

	20200116	20200117	20200118	20200119	20200120
20200109	9	51	7	5	7
20200110	12	28	6	7	41
20200111	11	53	8	5	45
20200112	68	82	7	13	41
20200113	163	95	54	43	9
20200114	418	189	81	119	13
20200115	*488	328	150	146	75
20200116	187	*470	388	216	81
20200117		175	*458	421	197
20200118			160	*358	338
20200119				137	*370
20200120					202

- 検索結果の並び順は “最もよくマッチする順” か “インデックスされた順” しか指定できず，実際に検索した結果を見ると，同じ検索語で検索した場合であっても検索結果の順位が固定されていなかった．
- 検索 API ではコミット日時を指定できない，かつ，検索結果にコミット日時が含まれないため，検索 API の返す結果のみでは，特定の日にコミットされた Dockerfile のみを収集して日ごとの傾向を見るような分析ができない．

そこで，同じ検索語で検索 API を繰り返し実行し，異なる Dockerfile が得られた場合にその Dockerfile を記録することで，1000 件以上の結果を収集する．すでに得られた Dockerfile のみしか得られなくなった場合，検索 API の実行を停止する．また，検索 API が返す Dockerfile が表示されるページへのリンク URL をたどり，そのページ内に含まれる Dockerfile のコミット日時を別途記録することで，いつコミットされた Dockerfile かどのタグを利用していたかを分析する．これにより，特定の日にコミットされた Dockerfile のみを収集し，日ごとの傾向を分析する．この方法では繰り返し検索 API を実行する必要があるため，この API に対して大きな負荷を与えてしまう可能性がある．GitHub REST API v3 にはレートリミットが設定されており，レートリミットの範囲内で検索 API を実行することで，大きな負荷を与えないよう配慮する．

Dockerfile を収集したのち，各 Dockerfile で利用されているタグを抽出する．各タグを抽出する際は，図 3 に示したベースイメージの記述方法をもとに，正規表現などを利用する．このとき，検索 API の結果に含まれている Dockerfile であっても，例えば，タグ部分が変数として定義されており，実行時にタグ部分を置換して実行するように作成された Dockerfil では，正規表現などによりタグが抽出できないため除外する．

上記の方法により，毎日日本時間で 17 時に検索 API を繰り返し実行し，ベースイメージに node を利用した Dockerfile を収集した結果を表 1 に示す．表の横方向は収集日を表しており，2 列目の “20200116” は 2020 年 1 月 16 日に収集した結果を表している．表の縦方向は Dockerfile のコミット日を表して

おり，2 行目の “20200109” は 2020 年 1 月 9 日にコミットされた Dockerfile を表している．つまり，2020 年 1 月 16 日に収集した結果では，2020 年 1 月 9 日にコミットされた node をベースイメージとする Dockerfile の数は 9 であったことを表している．

この結果から，検索 API ではコミット日が新しい Dockerfile は多く現れるが，コミット日が古くなるにつれて検索結果に現れる数が減少するという傾向があることがわかる．そこで，本手法では，取得数が最も多い前日にコミットされた Dockerfile のみを分析に利用することとした．表中の*印をつけた箇所がそれぞれの日にコミットされた Dockerfile として分析の対象とした Dockerfile 群を表している．

この方法では，Dockerfile を収集する時刻によっても収集結果が異なると考えられるが，あくまで日ごとのおおまかな傾向を確認することを目的としているため，ここでは議論しないものとする．

3.1.2 収集したタグの分析

次に，収集した Dockerfile から抽出したタグから傾向を分析する．1 つの Docker イメージに対して複数のタグを付与できるという Docker イメージの性質から，以下に示すように複数の分析方法が考えられる．

（1） 抽出したタグをそのまま利用し，日ごとに最も利用されている数が多いタグを分析する

（2） 抽出したタグをそのまま利用し，最も粒度が細かいタグのみを利用して，日ごとに最も利用されている数が多いタグを分析する

（3） 抽出したタグを同一イメージを指す最も粒度が細かいタグに変換したのち，日ごとに最も利用されている数が多いタグを分析する

（4） 抽出したタグをバージョンとタイプに分割し，バージョン部分に対して (1) の方法で分析する

（5） 抽出したタグをバージョンとタイプに分割し，バージョン部分に対して (2) の方法で分析する

それぞれの方法について，Dockerfile のコミット日時が図 2 の t_1 である場合を例として説明する．まず，(1) の方法では，取得したタグをそのまま集計することになるが，例えば “3” のように粒度が高いタグの利用数が最も多い場合，実際にはビルドするタイミングによって利用されるイメージが異なることから，(1) の方法ではどの Docker イメージが最も利用数が多いのか特定できない．(2) の方法では，例えば “3” というタグが抽出された場合，この Docker イメージに付与されている最も粒度が細かいタグが “3.7.0” であったとすると，“3” というタグは分析からは除外する．一方，(3) の方法の場合，“3.7.0” というタグが利用されたものとして分析する．これは，実際にはこのアプリケーションには “3” というバージョンは存在せず，バージョン 3.7.0 が含まれる Docker イメージがベースイメージとして利用されるためである．粒度が細かいタグ（セマンティックバージョニングを採用しているアプリケーションであればセマンティックバージョンを含むタグ）を利用することで，実際に利用されているバージョンを分析できると考えられる．

“3.8.2-buster” のように収集したタグにタイプが含まれる場合，(2) や (3) の方法では “3.8.2-buster” のまま集計されるが，(4) や (5) の方法では “3.8.2” として集計する．(2)，(3) の方法ではタイプとバージョン双方の傾向を分析できるが，(4)，(5) の方法ではバージョンごとの傾向を分析できる．本稿では，タイプごとに Docker イメージに不具合や脆弱性が含まれる場合については対象ではなく，バージョンごとの不具合や脆弱性に対応するためにアップデートする場合を対象としているため，(4) や (5) の方法が適していると考えられる．

一方で，(2)，(3) の方法のようにタイプまで含めて分類した場合，特に利用数の少ないタグの場合には傾向を分析するために十分なデータが取得できない可能性が考えられる．また，タイプごとの脆弱性について，詳しくは後述するが，“buster” のように Docker イメージ内に含まれるパッケージ数が多いタイプの場合，タイプを変更しない限り脆弱性への対応ができない場合も考えられるため，本稿では対象としないこととする．

3.2 脆弱性チェック手法

本節では，Docker イメージをダウンロードして脆弱性チェックをおこない，脆弱性が発見された場合には他のタグの Docker イメージに更新する手法について述べる．

Docker イメージの脆弱性をチェックする方法としては，Docker イメージに対応した脆弱性チェックツールを利用する方法が考えられる．具体的なツールとしては，Trivy[13]，Docker Bench for Security[14]，Clair[15]，Anchore Engine[16]などが挙げられる．他の方法としては，Docker Hub などのパブリックレジストリ上のサービスとして提供される脆弱性チェック機能を使う方法がある．

脆弱性チェックツールを利用する場合，対象の Docker イメージをダウンロードしたのち，脆弱性チェックツールを実行する必要がある．ユーザが Dockerfile をコミットし，コードリポジトリにプッシュした直後に Docker イメージがビルドされ，ベースイメージがダウンロードされると想定しているため，Dockerfile がコミットされた日時に実際に利用された Docker イメージのタグと，その Docker イメージの脆弱性チェックの結果が必要となる．また，図 2 に示したように，同一のタグが異なる Docker イメージに付与される場合もあるため，例えばユーザが Dockerfile をコミットした日時が t_1 の場合にはダイジェスト A，t_2 の場合にはダイジェスト B のイメージの脆弱性チェック結果を参照する必要がある．

過去の Docker イメージの脆弱性チェックの結果をあとから参照できるようにする方法として，以下に示す複数の方法が考えられる．

（1） 定期的に Docker Hub 上の Docker イメージをダウンロードし，脆弱性チェックをおこない，その結果を蓄積しておく．

---

13 : https://github.com/aquasecurity/trivy
14 : https://github.com/docker/docker-bench-security/
15 : https://github.com/quay/clair
16 : https://github.com/anchore/anchore-engine

（2） Docker Registry HTTP API V2[17]の情報を定期的に確認し，Docker イメージが更新されていれば，更新された Docker イメージのみをダウンロードして脆弱性チェックをおこない，その結果を蓄積しておく．

（3） (2) の方法に加え，一度ダウンロードした Docker イメージを保存しておき，脆弱性チェックに利用する定義ファイルが更新されるごとに再度チェックし直し，その結果も蓄積しておく．

従来，Docker Hub からの Docker イメージのダウンロードにはレートリミットは存在しなかったが，2020 年 11 月 20 日からレートリミットが導入された [18]ため，(2) や (3) の方法が望ましい．(1) の方法を利用する場合，収集頻度が更新頻度よりも低い場合，すべての Docker イメージの脆弱性をチェックできないという問題がある．また，Docker イメージが更新されたタイミングでは未知であった脆弱性があとから発見されるというケースも考えられる．そこで，(3) の方法により，過去に更新された Docker イメージに対しても脆弱性チェックを再度おこなうことで，そのような脆弱性情報を収集できる．しかし，(3) の方法では，一度ダウンロードした Docker イメージを保持し続ける必要があるため，巨大なストレージが必要となるという問題がある．

4 実　　験

本章では，提案手法によって Dockerfile におけるベースイメージのバージョンの更新タイミングを適切に推薦できるかどうかを考察するため，実験をおこなう．

4.1 実験環境

実験をおこなうにあたり，GitHub 上の Dockerfile と Docker Hub 上の Docker イメージの脆弱性情報を定期的に取得する．

Dockerfile については，3.1.1 節で述べた方法により，GitHub 上の Dockerfile を 1 日 1 回収集する．収集する対象の Docker イメージ名は，社内で利用しているユーザが多いと考えられる “golang”，“python”，“ruby” の 3 種類とした．収集期間は 2020 年 2 月 12 日から 2020 年 12 月 13 日で，毎日 17 時に収集を開始し，この順に収集をおこなった．つまり，2 番目に収集を開始する “python” 以降は，収集開始時刻は固定ではなく，前の Docker イメージの収集が終わったあとに順次開始した．集計時には，収集日の前日にコミットされたもののみを集計するため，日によって収集開始時刻が異なることによる影響は非常に少ないと考えている．収集した Dockerfile から，3.1.2 節で述べた (5) の方法により利用されているバージョンを推測し，集計をおこなう．

Docker イメージの脆弱性情報については，3.2 節で述べた (1) の方法により，Docker Hub 上の Docker イメージを週に 1 回ダウンロードし，Trivy を用いて脆弱性チェックをおこなった結果を収集する．収集する対象の Docker イメージ名は，Dockerfile を収集する Docker イメージと同様とした．この方法では週に 1 回以上更新された Docker イメージの脆弱性をチェックすることができないため，ここで取得した脆弱性チェックの結果には漏れがある可能性がある．ダウンロードは毎週土曜日の午前 2 時から順におこない，各イメージ名の全タグの Docker イメージを順にダウンロードして脆弱性チェックをおこない，ディスク容量の問題から脆弱性チェック後にダウンロードした Docker イメージは毎回削除した．

Dockerfile，Docker イメージの脆弱性情報いずれの収集においても，実験環境に用いたサーバのメンテナンスや GitHub，Docker Hub の障害等の理由により，データに抜けが存在する．提案手法では傾向を分析することで Dockerfile の更新を推薦することを目的としており，リアルタイムな更新は目的としていないため，本評価の結果には大きな影響はないと考えている．一方で，脆弱性が見つかった場合にはすぐに Dockerfile の更新を促すというアルゴリズムを組み込むことで，そうしたニーズにも答えられると考えられる．

4.2 実験結果

図 4，図 5，図 6 に，2020 年 2 月 12 日から 2020 年 12 月 13 日に収集した Dockerfile をもとに，それぞれ golang，python，ruby の Docker イメージにおける日ごとの各バージョンの利用率の変化を示す．

図 4 から，golang では最新版の利用率が目立って高くなっており，それ以外のバージョンの利用率は低いことがわかる．また，1.13 系列から 1.14 系列，1.14 系列から 1.15 系列へのマイナーバージョンの移行もスムーズにおこなわれていることがわかる．golang は比較的新しく登場した言語であり，利用者は積極的に新しいバージョンを利用していると考えられる．また，バージョン 1.14.5 やバージョン 1.14.7，バージョン 1.15.4 など，最も利用率が高かった日が数日しかないバージョンが存在する．このうち，バージョン 1.15.4 では，CVE-2020-28362, CVE-2020-28366, CVE-2020-28367 といった複数の脆弱性が報告されており，早期のアップデートが推奨されていた．しかし，バージョン 1.15.5 が公開された以降も脆弱性の存在するバージョンは少ないながらも利用されていることがわかる．これらの結果から，ユーザが利用していると考えられるベースイメージのバージョンと最も利用率の高いバージョンが異なる場合に，最も利用率が高いバージョンをユーザに対して推薦することは有用であると考えられる．

一方で，バージョン 1.15.4 以降の脆弱性チェックの結果を表 2 に示す．Trivy では，検出された脆弱性は深刻度が高い順に CRITICAL，HIGH，MEDIUM，LOW に分類され，深刻度がわからないものは UNKNOWN に分類される．この表では，それぞれ alpine タイプの脆弱性チェックの結果を示している．これは，他の OS では含まれるパッケージなどが多く，恒常的になんらかの脆弱性が見つかっているため，golang のバージョンの違いによる脆弱性チェックの結果の比較が難しいからである．表から，前述したバージョン 1.15.4 までに存在していた脆弱性は Trivy では検出されていないことがわかる．一方で，

17 : https://docs.docker.com/registry/spec/api/
18 : https://www.docker.com/increase-rate-limits

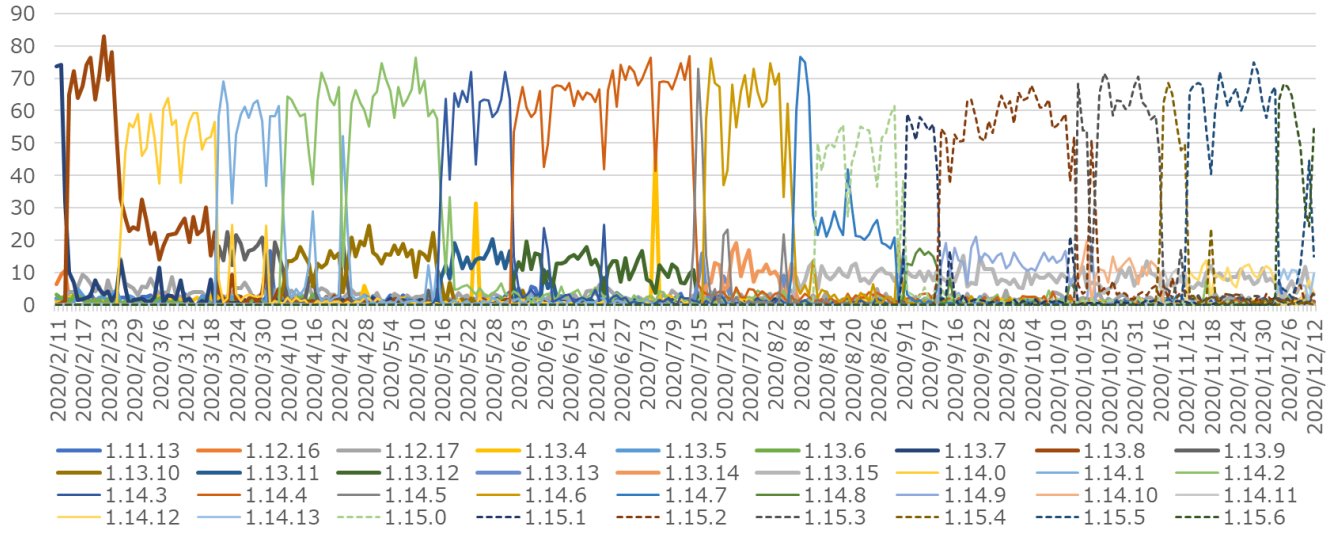

図 4　golang イメージにおける各バージョンの利用率

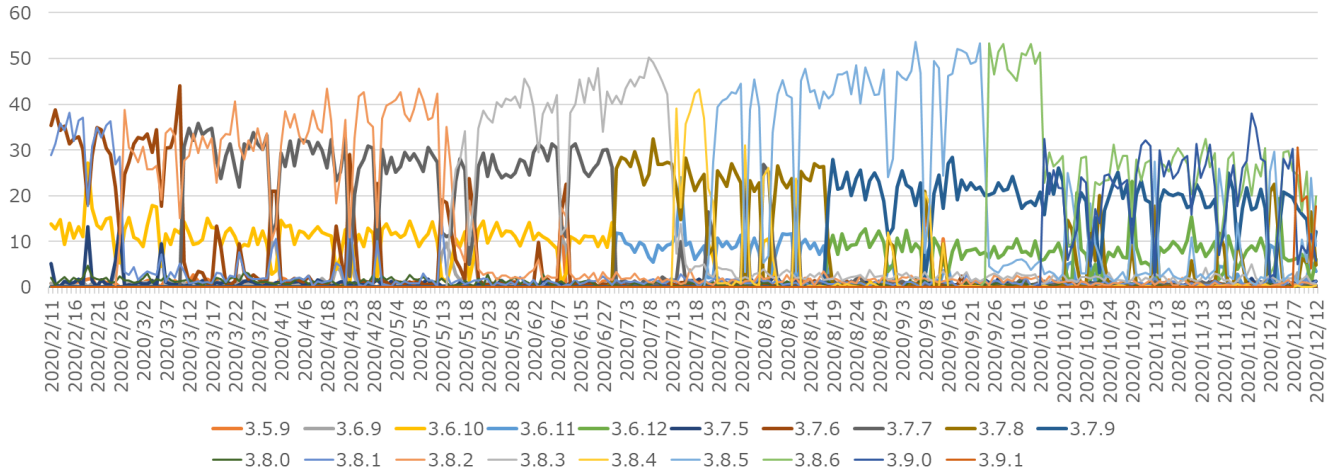

図 5　python イメージにおける各バージョンの利用率

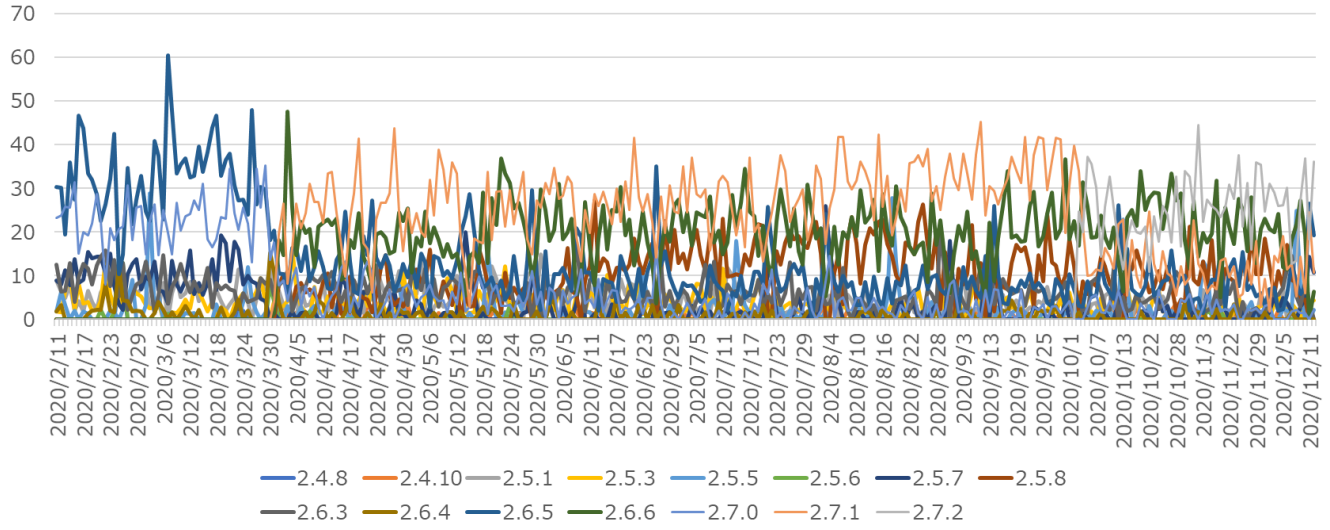

図 6　ruby イメージにおける各バージョンの利用率

バージョン 1.15.6 では，2020 年 12 月 4 日の時点では脆弱性が存在しているが，2020 年 12 月 11 日の時点では脆弱性が解消されている．これは，OS に含まれていた脆弱性を解消するために同一のタグで新たな Docker イメージが公開されたため

表 2　golang イメージにおける Trivy による脆弱性チェック結果

チェック日	バージョン 1.15.4	バージョン 1.15.5	バージョン 1.15.6
20201113	脆弱性なし	脆弱性なし	-
20201120	UNKNOWN 1 件	UNKNOWN 1 件	-
20201127	UNKNOWN 1 件	UNKNOWN 1 件	-
20201204	MEDIUM 1 件	MEDIUM 1 件	MEDIUM 1 件
20201211	HIGH 3 件，MEDIUM 1 件	HIGH 3 件，MEDIUM 1 件	脆弱性なし

表 3　python イメージにおける Trivy による脆弱性チェック結果

チェック日	バージョン 3.7.8	バージョン 3.7.9
20200815	脆弱性なし	-
20200822	脆弱性なし	脆弱性なし
20200829	MEDIUM 1 件	MEDIUM 1 件
20200904	MEDIUM 1 件	MEDIUM 1 件
20200815	HIGH 1 件 UNKNOWN 2 件	HIGH 1 件 UNKNOWN 2 件

である．この結果から，利用率が最も高いバージョンをユーザに対して推薦するだけでなく，脆弱性情報を併用し，ベースイメージが更新されたタイミングで新たなベースイメージを利用して再度 Dockerfile をビルドするようにユーザに対して推薦することが有用であると考えられる．

図 5 から，python ではそれぞれのマイナーバージョンにおける最新版の利用率が高い傾向にあることがわかる．また，3.9 系列がリリースされた以降では，3.9 系列と 3.8 系列がほぼ同数利用されている．この結果から，3.9 系列へのマイナーバージョンのアップデートを躊躇しているユーザが多いと考えられる．バージョン 3.7.8 では，深刻度の非常に高い脆弱性である CVE-2020-15801 を含む 4 件の脆弱性が見つかっており，修正したバージョン 3.7.9 がリリースされている．バージョン 3.7.8 以降の alpine タイプの脆弱性チェックの結果を表 3 に示す．表から，バージョン 3.7.8 と 3.7.9 の脆弱性チェック結果が同じになっていることがわかる．Trivy は python におけるバージョンロックファイルである Pipfile.lock や poetry.lock をスキャンし，ライブラリの脆弱性チェックをおこなう機能をもっているが，python 自体の脆弱性はチェックされていないか，本脆弱性に関する情報が登録されていないためにこのような結果になったと考えられる．提案手法では，利用率の高いバージョンへの変更を推薦することにより，脆弱性チェックツールで検出できない脆弱性がある場合でもアップデートを推薦できると考えられる．

図 6 から，ruby では 2.6 系列と 2.7 系列がほぼ同数利用されている状況が続いており，同様に 2.7 系列へのマイナーバージョンのアップデートを躊躇しているユーザが多いと考えられる．このような場合，数日続けて同じバージョンの利用率が最も高くなるまで，マイナーバージョンのアップデートをユーザに推薦しないほうがよいと考えられる．

5 ま と め

本稿では，GitHub 上で公開されている Dockerfile から，どのバージョンのベースイメージが最も利用率が高いかを推測し，最も利用率が高いバージョンへのアップデートを推薦する手法と，Docker Hub 上で公開されている Docker イメージを定期的にダウンロードし，Trivy を用いて脆弱性をチェックすることで脆弱性情報を収集し，脆弱性が修正された際に修正済みのバージョンへアップデートする手法を提案した．提案手法の実現性および性能を検証するための実験をおこない，提案手法によりユーザに対して適切な推薦をおこなう見通しを立てた．

今後の課題としては，以下の項目が挙げられる．

- ユーザに対してバージョン変更を推薦するタイミングの検討
- 利用率および脆弱性情報を両立してユーザに対して推薦する手法の提案
- ユーザのポリシーに応じたタグ推薦手法の提案

文　献

[1] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi and H. C. Gall, “An Empirical Analysis of the Docker Container Ecosystem on GitHub,” in Proc. IEEE/ACM 14th International Conference on Mining Software Repositories (MSR 2017), pp. 323–333, May 2017.

[2] J. Dietrich, D. Pearce, J. Stringer, A. Tahir and K. Blincoe, “Dependency Versioning in the Wild,” in Proc. IEEE/ACM 16th International Conference on Mining Software Repositories (MSR 2019), 2019, pp. 349–359, May 2019.

[3] Z. Huang, S. Wu, S. Jiang and H. Jin, “FastBuild: Accelerating Docker Image Building for Efficient Development and Deployment of Container,” in Proc. 35th Symposium on Mass Storage Systems and Technologies (MSST 2019), pp. 28–37, May 2019.

[4] S. Kitajima and A. Sekiguchi, “Latest Image Recommendation Method for Automatic Base Image Update in Dockerfile”, in Proc. 18th International Conference on Service-Oriented Computing (ICSOC 2020), pp. 547–562, Dec. 2020.

[5] S. Raemaekers, A. van Deursen and J. Visser, “Semantic Versioning versus Breaking Changes: A Study of the Maven Repository,” in Proc. IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM 2014), pp. 215–224, Sep. 2014.

[6] J. Shah, D. Dubaria and J. Widhalm, “A Survey of DevOps Tools for Networking,” in Proc. 9th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (IEEE UEMCON 2018), pp. 185–188, Nov. 2018.

[7] Y. Zhang, G. Yin, T. Wang, Y. Yu and H. Wang, "An Insight Into the Impact of Dockerfile Evolutionary Trajectories on Quality and Latency," in Proc. IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC 2018), pp. 138–143, June 2018.