Learning from, Understanding, and Supporting DevOps Artifacts for Docker

Jordan Henkel
University of Wisconsin–Madison, USA
jjhenkel@cs.wisc.edu

Christian Bird
Microsoft Research, USA
Christian.Bird@microsoft.com

Shuvendu K. Lahiri
Microsoft Research, USA
Shuvendu.Lahiri@microsoft.com

Thomas Reps
University of Wisconsin–Madison, USA
reps@cs.wisc.edu

ABSTRACT

With the growing use of DevOps tools and frameworks, there is an increased need for tools and techniques that support *more than code*. The current state-of-the-art in static developer assistance for tools like Docker is limited to shallow syntactic validation. We identify three core challenges in the realm of learning from, understanding, and supporting developers writing DevOps artifacts: (i) nested languages in DevOps artifacts, (ii) rule mining, and (iii) the lack of semantic rule-based analysis. To address these challenges we introduce a toolset, `binnacle`, that enabled us to ingest 900,000 GitHub repositories.

Focusing on Docker, we extracted approximately 178,000 unique Dockerfiles, and also identified a Gold Set of Dockerfiles written by Docker experts. We addressed challenge (i) by reducing the number of effectively uninterpretable nodes in our ASTs by over 80% via a technique we call *phased parsing*. To address challenge (ii), we introduced a novel rule-mining technique capable of recovering two-thirds of the rules in a benchmark we curated. Through this automated mining, we were able to recover 16 new rules that were not found during manual rule collection. To address challenge (iii), we manually collected a set of rules for Dockerfiles from commits to the files in the Gold Set. These rules encapsulate best practices, avoid docker build failures, and improve image size and build latency. We created an analyzer that used these rules, and found that, on average, Dockerfiles on GitHub violated the rules *five times more frequently* than the Dockerfiles in our Gold Set. We also found that industrial Dockerfiles fared no better than those sourced from GitHub.

The learned rules and analyzer in `binnacle` can be used to aid developers in the IDE when creating Dockerfiles, and in a post-hoc fashion to identify issues in, and to improve, existing Dockerfiles.

CCS CONCEPTS

• **Software and its engineering → Empirical software validation**; General programming languages; • **Theory of computation** → Program semantics; *Abstraction*; • **Information systems →** *Data mining*.

KEYWORDS

Docker, DevOps, Mining, Static Checking

ACM Reference Format:
Jordan Henkel, Christian Bird, Shuvendu K. Lahiri, and Thomas Reps. 2020. Learning from, Understanding, and Supporting DevOps Artifacts for Docker. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380406

1 INTRODUCTION

With the continued growth and rapid iteration of software, an increasing amount of attention is being placed on services and infrastructure to enable developers to test, deploy, and scale their applications quickly. This situation has given rise to the practice of *DevOps*, a blend of the words *Development* and *Operations*, which seeks to build a bridge between both practices, including deploying, managing, and supporting a software system [23]. Bass *et al.* define DevOps as, the "set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality" [11]. DevOps activities include building, testing, packaging, releasing, configuring, and monitoring software. To aid developers in these processes, tools such as TravisCI [9], CircleCI [1], Docker [2], and Kubernetes [6], have become an integral part of the daily workflow of thousands of developers. Much has been written about DevOps (see, for example, [16] and [22]) and various practices of DevOps have been studied extensively [20, 27, 31, 31–33, 40].

DevOps tools exist in a heterogenous and rapidly evolving landscape. As software systems continue to grow in scale and complexity, so do DevOps tools. Part of this increase in complexity can be seen in the input formats of DevOps tools: many tools, like Docker [1], Jenkins [4], and Terraform [8], have custom DSLs to describe their input formats. We refer to such input files as *DevOps artifacts*.

Historically, DevOps artifacts have been somewhat neglected in terms of industrial and academic research (though they have received interest in recent years [28]). They are not "traditional" code, and therefore out of the reach of various efforts in automatic mining and analysis, but at the same time, these artifacts are *complex*. Our discussions with developers tasked with working on these artifacts indicate that they learn just enough to "get the job done."

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380406

Phillips *et al.* found that there is little perceived benefit in becoming an expert, because developers working on builds told them "if you are good, no one ever knows about it [26]." However, there is a strong interest in tools to assist the development of DevOps artifacts: even with its relatively shallow syntactic support, the VS Code Docker extension has over 3.7 million unique installations [24]. Unfortunately, the availability of such a tool has not translated into the adoption of best practices. We find that, on average, Dockerfiles on GitHub have nearly five times as many rule violations as those written by Docker experts. These rule violations, which we describe in more detail in §4, range from true bugs (such as simply forgetting the `-y` flag when using `apt-get install` which causes the build to hang) to violations of community established best practices (such as forgetting to use `apk add`'s `–no-cache` flag).

The goal of our work is as follows:

> We seek to address the need for more effective semantics-aware tooling in the realm of DevOps artifacts, with the ultimate goal of reducing the gap in quality between artifacts written by experts and artifacts found in open-source repositories.

We have observed that best practices for tools like Docker have arisen, but engineers are often unaware of these practices, and therefore unable to follow them. Failing to follow these best practices can cause longer build times and larger Docker images at best, and eventual broken builds at worst. To ameliorate this problem, we introduce `binnacle`: the first toolset for semantics-aware rule mining from, and rule enforcement in, Dockerfiles. We selected Dockerfiles as the initial type of artifact because it is the most prevalent DevOps artifact in industry (some 79% of IT companies use it [27]), has become the de-facto container technology in OSS [15, 38], and it has a characteristic that we observe in many other types of DevOps artifacts, namely, fragments of shell code are embedded within its declarative structure.

Because many developers are comfortable with the Bash shell in an interactive context, they may be unaware of the differences and assumptions of shell code in the context of DevOps tools. For example, many bash tools use a caching mechanism for efficiency. Relying on and not removing the cache can lead to wasted space, outdated packages or data, and in some cases, broken builds. Consequently, one must always invoke `apt-get update` before installing packages, and one should also delete the cache after installation. Default options for commands may need to be overridden often in a Docker setting. For instance, users almost always want to install recommended dependencies. However, using recommended dependencies (which may change over time in the external environment of apt package lists) can silently break future Dockerfile builds and, in the near term, create a likely wastage of space, as well as the possibility of implicit dependencies (hence the need to use the `–no-recommends` option). Thus, a developer who may be considered a Bash or Linux expert can still run afoul of Docker Bash pitfalls.

To create the `binnacle` toolset, we had to address three challenges associated with DevOps artifacts: (C1) the challenge of nested languages (e.g., arbitrary shell code is embedded in various parts of the artifact), (C2) the challenge of rule encoding and automated rule mining, and (C3) the challenge of static rule enforcement. As a prerequisite to our analysis and experimentation, we also collected approximately 900,000 GitHub repositories, and from these repositories, captured approximately 219,000 Dockerfiles (of which 178,000 are unique). Within this large corpus of Dockerfiles, we identified a subset written by Docker experts: this *Gold Set* is a collection of high-quality Dockerfiles that our techniques use as an oracle for Docker best practices.[1]

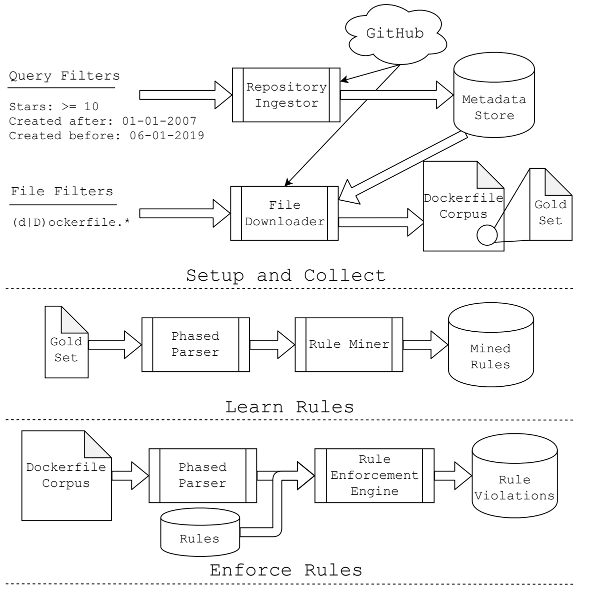


Fig. 1: An overview of the `binnacle` toolset.

To address (C1), we introduced a novel technique for generating structured representations of DevOps artifacts in the presence of nested languages, which we call *phased parsing*. By observing that there are a relatively small number of commonly used command-line tools—and that each of these tools has easily accessible documentation (via manual/help pages)—we were able to enrich our DevOps ASTs and reduce the percentage of *effectively uninterpretable* leaves (defined in §3.1) in the ASTs by over 80%.

For the challenge of rule encoding and rule mining (C2), we took a three-pronged approach:

(1) We introduced Tree Association Rules (TARs), and created a corpus of *Gold Rules* manually extracted from changes made to Dockerfiles by Docker experts (§3.2).
(2) We built an automated rule miner based on frequent sub-tree mining (§3.4).
(3) We performed a study of the quality of the automatically mined rules using the the *Gold Rules* as our ground-truth benchmark (§4.2).

In seminal work by Sidhu et al. [30], they attempted to learn rules to aid developers in creating DevOps artifacts, specifically Travis CI files. They concluded that their "vision of a tool that provides suggestions to build CI specifications based on popular sequences of phases and commands cannot be realized." In our work, we adopt their vision, and show that it is indeed achievable. There is a simple explanation for why our results differ from theirs. In our work,

[1] Data avaliable at: https://github.com/jjhenkel/binnacle-icse2020

we use our phased parser to go two levels deep in a hierarchy of nested languages, whereas Sidhu et al. only considered one level of nested languages. Moreover, when we mine rules, we mine them starting with the *deepest* level of language nesting. Thus, our rules are mined from the results of a layer of parsing that Sidhu et al. did not perform, and they are mined *only* from that layer.

Finally, to address (C3), the challenge of static rule enforcement, we implemented a static enforcement engine that takes, as input, Tree Association Rules (TARs). We find that Dockerfiles on GitHub are nearly five times worse (with respect to rule violations) when compared to Dockerfiles written by experts, and that Dockerfiles collected from industry sources are no better. This gap in quality is precisely what the `binnacle` toolset seeks to address.

In summary, we make four core contributions:

(1) A dataset of 178,000 unique Dockerfiles, processed by our phased parser, harvested from *every public GitHub repository with 10 or more stars*,[2] and a toolset, called `binnacle`, capable of ingesting and storing DevOps artifacts.

(2) A technique for addressing the nested languages in DevOps artifacts that we call *phased parsing*.

(3) An automatic rule miner, based on frequent sub-tree mining, that produces rules encoded as Tree Association Rules (TARs).

(4) A static rule-enforcement engine that takes, as input, a Dockerfile and a set of TARs and produces a listing of rule violations.

For the purpose of evaluation, we provide experimental results against Dockerfiles, but, in general, the techniques we describe in this work are applicable to any DevOps artifact with nested shell (e.g., Travis CI and Circle CI). The only additional component that `binnacle` requires to operate on a new artifact type is a top-level parser capable of identifying any instances of embedded shell. Given such a top-level parser, the rest of the `binnacle` toolset can be applied to learn rules and detect violations.

Our aim is to provide help to developers in various activities. As such, `binnacle`'s rule engine can be used to aid developers when writing/modifying DevOps artifacts in an IDE, to inspect pull requests, or to improve existing artifacts already checked in and in use.

2 DATA ACQUISITION

A prerequisite to analyzing and learning from DevOps artifacts is gathering a large sample of representative data. There are two challenges we must address with respect to data acquisition: (D1) the challenge of gathering *enough* data to do interesting analysis, and (D2) the challenge of gathering *high-quality* data from which we can mine rules. To address the first challenge, we created the `binnacle` toolset: a dockerized distributed system capable of ingesting a large number of DevOps artifacts from a configurable selection of GitHub repositories. `binnacle` uses a combination of Docker and Apache Kafka to enable dynamic scaling of resources when ingesting a large number of artifacts. Fig. 1 gives an overview of the three primary tools provided by the `binnacle` toolset: a tool for data acquisition, which we discuss in this section, a tool for rule learning (discussed further in §3.4), and a tool for static rule enforcement (discussed further in §3.5).

Although the architecture of `binnacle` is interesting in its own right, we refer the reader to the `binnacle` GitHub repository for more details.[3] For the remainder of this section, we instead describe the data we collected using `binnacle`, and our approach to challenge (D2): the need for *high-quality* data.

Using `binnacle`, we ingested *every public repository on GitHub with ten or more stars*. This process yielded approximately 900,000 GitHub repositories. For each of these 900,000 repositories, we gathered a listing of all the files present in each repository. This listing of files was generated by looking at the `HEAD` of the default branch for each repository. Together, the metadata and file listings for each repository were stored in a database. We ran a script against this database to identify the files that were likely Dockerfiles using a permissive filename-based filter. This process identified approximately 240,000 likely Dockerfiles. Of those 240,000 likely Dockerfiles, only 219,000 were successfully downloaded and parsed as Dockerfiles. Of the 219,000 remaining files, approximately 178,000 were unique based on their SHA1 hash. It is this set, of approximately 178,000 Dockerfiles, that we will refer to as our corpus of Dockerfiles.

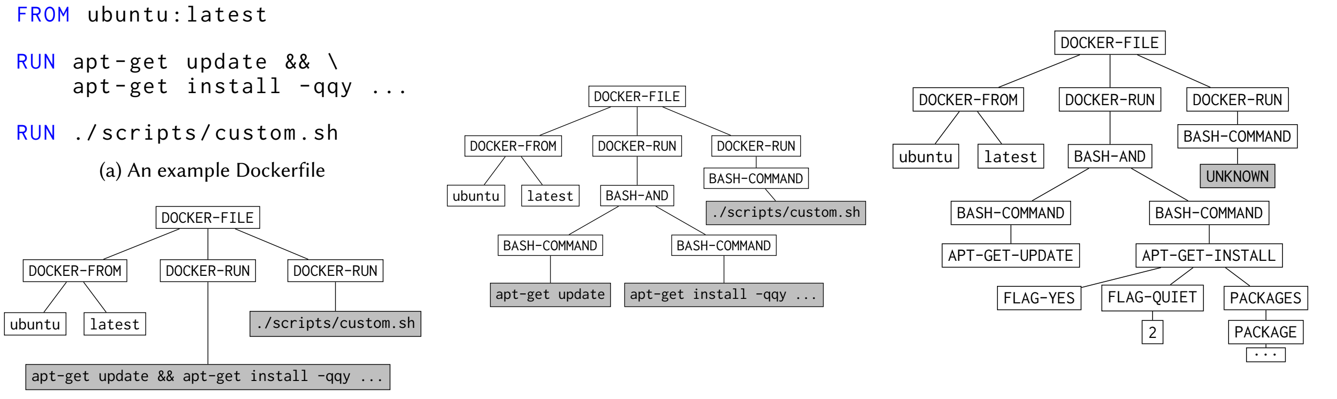
Although both the number of repositories we ingested and the number of Dockerfiles we collected were large, we still had not addressed challenge (D2): high-quality data. To find high-quality data, we looked within our Dockerfile corpus and extracted every Dockerfile that originally came from the `docker-library/` GitHub organization. This organization is run by Docker, and houses a set of official Dockerfiles written by and maintained by Docker experts. There are approximately 400 such files in our Dockerfile corpus. We will refer to this smaller subset of Dockerfiles as the *Gold Set*. Because these files are Dockerfiles created and maintained by Docker's own experts, they presumably represent a higher standard of quality than those produced by non-experts. This set provides us with a solution to challenge (D2)—the Gold Set can be used as an oracle for good Dockerfile hygiene. In addition to the Gold Set, we also collected approximately 5,000 Dockerfiles from several industrial repositories, with the hope that these files would also be a source of high-quality data.

3 APPROACH

The `binnacle` toolset, shown in Fig. 1, can be used to ingest large amounts of data from GitHub. This capability is of general use to anyone looking to analyze GitHub data. In this section, we describe the three core contributions of our work: phased parsing, rule mining, and rule enforcement. Each of these contributions is backed by a corresponding tool in the `binnacle` toolset: (i) *phased parsing* is enabled by `binnacle`'s phased parser (shown in the Learn Rules and Enforce Rules sections of Fig. 1); (ii) *rule mining* is enabled by `binnacle`'s novel frequent-sub-tree-based rule miner (shown in the Learn Rules section of Fig. 1); and *rule enforcement* is provided by `binnacle`'s static rule-enforcement engine (shown in the Enforce Rules section of Fig. 1). Each of these three tools and contributions was inspired by one of the three challenges we identified in the realm of learning from and understating DevOps artifacts (nested languages, prior work that identifies rule mining as unachievable

[2] We selected repositories created after January 1st, 2007 and before June 1st, 2019.

[3] https://github.com/jjhenkel/binnacle-icse2020



```
FROM ubuntu:latest

RUN apt-get update && \
    apt-get install -qqy ...

RUN ./scripts/custom.sh
```

(a) An example Dockerfile

(b) Phase I: Top-level parsing is performed

(c) Phase II: Embedded bash is parsed

(d) Phase III: The AST is enriched with the results of parsing the top-50 commands

Fig. 2: An example Dockerfile at each of the three phases of our phased-parsing technique (gray nodes are *effectively uninterpretable (EU)*)

[30], and static rule enforcement). Together, these contributions combine to create the binnacle toolset: the first structure-aware automatic rule miner and enforcement engine for Dockerfiles (and DevOps artifacts, in general).

3.1 Phased Parsing

One challenging aspect of DevOps artifacts in general (and Dockerfiles in particular) is the prevalence of nested languages. Many DevOps artifacts have a top-level syntax that is simple and declarative (JSON, Yaml, and XML are popular choices). This top-level syntax, albeit simple, usually allows for some form of embedded scripting. Most commonly, these embedded scripts are bash. Further complicating matters is the fact that bash scripts usually reference common command-line tools, such as apt-get and git. Some popular command-line tools, like python and php, may even allow for further nesting of languages. Other tools, like GNU's find, allow for more bash to be embedded as an argument to the command. This complex nesting of different languages creates a challenge: how do we represent DevOps artifacts in a structured way?

Previous approaches to understanding and analyzing DevOps artifacts have either ignored the problem of nested languages, or only addressed one level of nesting (the embedded shell within the top-level format) [17, 30]. We address the challenge of structured representations in a new way: we employ *phased parsing* to progressively enrich the AST created by an initial top-level parse. Fig. 2 gives an example of *phased parsing*—note how, in Fig. 2(b), we have a shallow representation given to us by a simple top-level parse of the example Dockerfile. After this first phase, almost all of the interesting information is wrapped up in leaf nodes that are string literals. We call such nodes *effectively uninterpretable* (EU) because we have no way of reasoning about their contents. These literal nodes, which have further interesting structure, are shown in gray. After the second phase, shown in Fig. 2(c), we have enriched the structured representation from Phase I by parsing the embedded bash. This second phase of parsing further refines the AST constructed for the example, but, somewhat counterintuitively, this refinement also introduces even more literal nodes with undiscovered structure. Finally, the third phase of parsing enriches the AST by parsing the options "languages" of popular command-line tools (see Fig. 2(d)). By parsing within these command-line languages, we create a representation of DevOps artifacts that contains *more structured information* than competing approaches.

To create our phased parser we leverage the following observations:

1. There are a small number of commonly used command-line tools. Supporting the top-50 most frequently used tools allows us to cover over 80% of command-line-tool invocations in our corpus.
2. Popular command-line tools have documented options. This documentation is easily accessible via manual pages or some form of embedded help.

Because of observation (1), we can focus our attention on the most popular command-line tools, which makes the problem of phased parsing tractable. Instead of somehow supporting all possible embedded command-line-tool invocations, we can, instead, provide support for the top-N commands (where N is determined by the amount of effort we are willing to expend). To make this process uniform and simple, we created a parser generator that takes, as input, a declarative schema for the options language of the command-line tool of interest. From this schema, the parser generator outputs a parser that can be used to enrich the ASTs during Phase III of parsing. The use of a parser generator was inspired by observation (2): the information available in manual pages and embedded help, although free-form English text, closely corresponds to the schema we provide our parser generator. This correspondence is intentional. To support more command-line tools, one merely needs to identify appropriate documentation and transliterate it into the schema format we support. In practice, creating the schema for a typical command-line tool took us between 15 and 30 minutes. Although the parser generator is an integral and interesting piece of infrastructure, we forego a detailed description of the input schema the generator requires and the process of transliterating manual pages; instead, we now present the rule-encoding scheme that binnacle uses both for rule enforcement and rule mining.

3.2 Tree Association Rules (TARs)

The second challenge the binnacle toolset seeks to address (rule encoding) is motivated by the need for both automated rule mining

Table 1: Detailed breakdown of the *Gold Rules*. (All rules are listed; the rules that passed confidence/support filtering, described in §3.5, are shaded.)

Rule Name	Bash Best-practice?	Immediate Violation Consequences	Future Violation Consequences	Gold Set Support	Gold Set Confidence
pipUseCacheDir	No	Space wastage	Increased attack surface	15	46.67%
npmCacheCleanUseForce	No	Space wastage	Increased attack surface	14	57.14%
mkdirUsrSrcThenRemove	Yes	Space wastage	Increased attack surface	129	68.99%
rmRecurisveAfterMktempD	Yes	Space wastage	Increased attack surface	632	77.22%
curlUseFlagF	No	None	Easier to add future bugs	72	77.78%
tarSomethingRmTheSomething	Yes	Space wastage	Increased attack surface	209	88.52%
apkAddUseNoCache	No	Space wastage	Increased attack surface	250	89.20%
aptGetInstallUseNoRec	No	Space wastage	Build failure	525	90.67%
curlUseHttpsUrl	Yes	Insecure	Insecure	57	92.98%
gpgUseBatchFlag	No	Build reliability	Build reliability	455	94.51%
sha256sumEchoOneSpace	Yes	Build failure	N/A	132	95.45%
gpgUseHaPools	No	Build reliability	Build reliability	205	97.07%
configureUseBuildFlag	No	None	Easier to add future bugs	128	98.44%
wgetUseHttpsUrl	Yes	Insecure	Insecure	290	98.97%
aptGetInstallRmAptLists	No	Space wastage	Increased attack surface	525	99.43%
aptGetInstallUseY	No	Build failure	N/A	525	100.00%
aptGetUpdatePrecedesInstall	No	Build failure	N/A	525	100.00%
gpgVerifyAscRmAsc	Yes	Space wastage	Increased attack surface	172	100.00%
npmCacheCleanAfterInstall	No	Space wastage	Increased attack surface	12	100.00%
gemUpdateSystemRmRootGem	No	Space wastage	Increased attack surface	11	100.00%
gemUpdateNoDocument	No	Space wastage	Increased attack surface	11	100.00%
yumInstallForceYes	No	Build failure	N/A	3	100.00%
yumInstallRmVarCacheYum	No	Space wastage	Increased attack surface	3	100.00%

Precedes

$$\frac{\texttt{(APT-GET-INSTALL)}}{\texttt{(APT-GET-UPDATE)}}$$

(a) Intuitively, this rule states that an `apt-get install` must be preceded (in the same layer of the Dockerfile) by an `apt-get update`.

Follows

$$\frac{\texttt{(APT-GET-INSTALL)}}{\texttt{(RM (RM-F-RECURSIVE) (RM-PATH (ABS-APT-LISTS)))}}$$

(b) Intuitively, this rule states that a certain directory must be removed (in the same layer of the Dockerfile) following an `apt-get install`.

Child-Of

$$\frac{\texttt{(APT-GET-INSTALL [*])}}{\texttt{(FLAG-NO-RECOMMENDS)}}$$

(c) Here, the user must select where, in the antecedent subtree, to bind a region to search for the consequent. This binding is represented by the `[*]` marker.

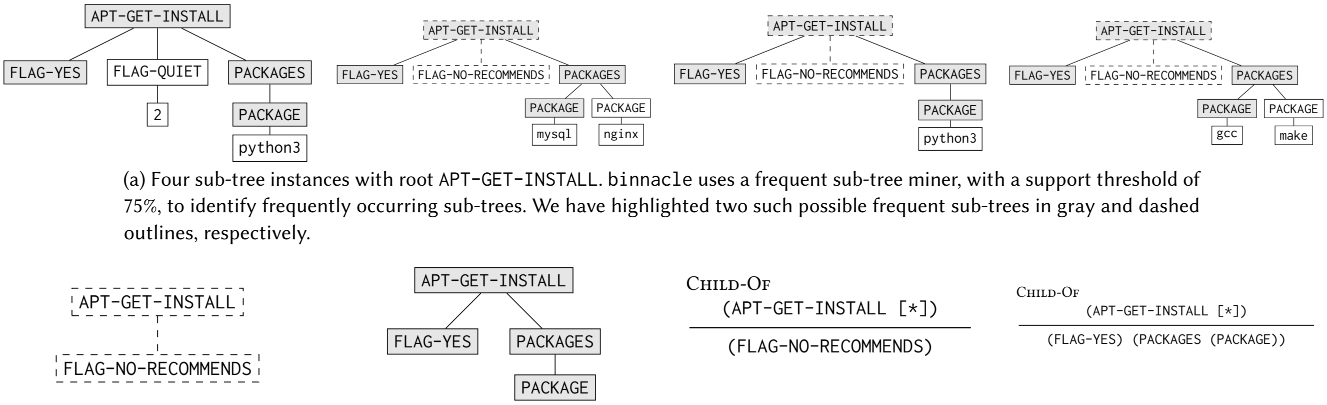
Fig. 3: Three example Tree Association Rules (TARs). Each TAR has, above the bar, an antecedent subtree encoded as an S-expression and, below the bar, a consequent subtree encoded in the same way.

and static rule enforcement. In both applications, there needs to be a consistent and powerful encoding of expressive rules with straightforward syntax and clear semantics. As part of developing this encoding, we curated a set of *Gold Rules* and wrote a rule-enforcement engine capable of detecting violations of these rules. We describe this enforcement engine in greater detail in §3.5. To create the set of *Gold Rules*, we returned to the data in our Gold Set of Dockerfiles. These Dockerfiles were obtained from the `docker-library/` organization on GitHub. We manually reviewed merged pull requests to the repositories in this organization. From the merged pull requests, if we thought that a change was applying a best practice or a fix, we manually formulated, as English prose, a description of the change. This process gave us approximately 50 examples of *concrete changes made by Docker experts*, paired with descriptions of the general pattern being applied.

From these concrete examples, we devised 23 rules. A summary of these rules is given in Table 1. Most examples that we saw could be framed as association rules of some form. As an example, a rule may dictate that using `apt-get install ...` requires a preceding `apt-get update`. Rules of this form can be phrased in terms of an antecedent and consequent. The only wrinkle in this simple approach is that both the antecedent and the consequent are sub-trees of the tree representation of Dockerfiles. To deal with tree-structured data, we specify two pieces of information that helps restrict *where* the consequent can occur in the tree, relative to the antecedent:

(1) Its location: the consequent can either (i) *precede* the antecedent, (ii) *follow* the antecedent, or (iii) *be a child of* the antecedent in the tree.

(2) Its scope: the consequent can either be (i) in the *same piece* of embedded shell as the antecedent (intra-directive), or (ii) it can be allowed to exist in a *separate piece* of embedded shell (inter-directive). Although we can encode and enforce inter-directive rules, our miner is only capable of returning intra-directive rules (as explained in §3.4). Therefore, all of the rules we show have an intra-directive scope.



(a) Four sub-tree instances with root APT-GET-INSTALL. binnacle uses a frequent sub-tree miner, with a support threshold of 75%, to identify frequently occurring sub-trees. We have highlighted two such possible frequent sub-trees in gray and dashed outlines, respectively.

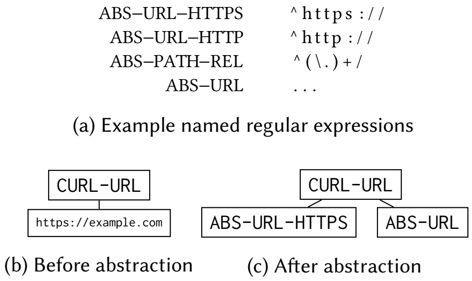
(b) The two frequently occuring sub-trees extracted from the example input corpus in Fig. 4(a); these trees become likely consequents.

(c) Tree Association Rules created automatically from the likely consequents in Fig. 4(b). The antecedent denotes the set of all sub-trees with the indicated root node-type.

Fig. 4: A depiction of rule mining in binnacle via frequent sub-tree mining.

From an antecedent, a consequent, and these two pieces of localizing information, we can form a complete rule against which the enriched ASTs created by the phased parser can be checked. We call these Tree Association Rules (TARs). Three example TARs are given in Fig. 3. We are not the first to propose Tree Association Rules; Mazuran et al. [25] proposed TARs in the context of extracting knowledge from XML documents. The key difference is that their TARs require that the consequent be a child of the antecedent in the tree, while we allow for the consequent to occur outside of the antecedent, either preceding it or succeeding it. Although we allow for this more general definition of TARs, our miner is only capable of mining *local* TARs—that is, TARs in the style of Mazuran et al. [25]; however, our static rule-enforcement engine has no such limitation.

Rule impacts. For each of the Gold rules, Table 1 provides the consequences of a rule violation and a judgement as to whether a given rule is unique to Dockerfiles or more aligned with general Bash best-practices. In general, we note that rule violations have varying consequences, including space wastage, container bloat (and consequent increased attack surface), and instances of outright build failure. Additionally, two-thirds of the Gold rules are *unique to using Bash in the context of a Dockerfile.*



ABS–URL–HTTPS	^https://
ABS–URL–HTTP	^http://
ABS–PATH–REL	^(\.)+/
ABS–URL	...

(a) Example named regular expressions

(b) Before abstraction (c) After abstraction

Fig. 5: An example of the abstraction process.

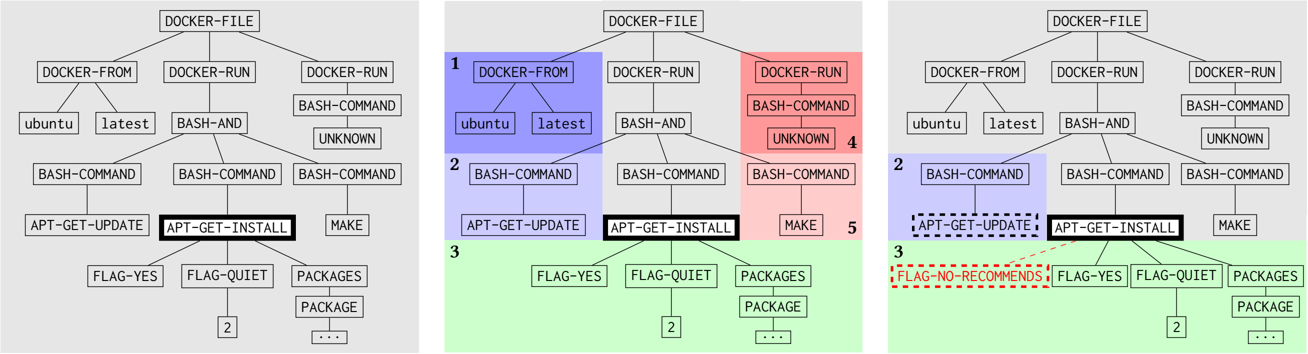
3.3 Abstraction

binnacle's rule miner and static rule-enforcement engine both employ an *abstraction* process. The abstraction process is complementary to *phased parsing*—there may still be information within literal values even when those literals are not from some well-defined sub-language. During the abstraction process, for each tree in the input corpus, every literal value residing in the tree is removed, fed to an abstraction subroutine, and replaced by either zero, one, or several abstract nodes (these abstract nodes are produced by the abstraction subroutine). The abstraction subroutine simply applies a user-defined list of named regular expressions to the input literal value. For every matched regular expression, the abstraction subroutine returns an abstract node whose type is set to the name of the matched expression. For example, one abstraction we use attempts to detect URLs; another detects if the literal value is a Unix path and, if so, whether it is relative or absolute. The abstraction process is depicted in Fig. 5. The reason for these abstractions is to help both binnacle's rule-learning and static-rule-enforcement phases by giving these tools the vocabulary necessary to reason about properties of interest.

3.4 Rule Mining

The binnacle toolset approaches rule mining by, first, focusing on a specific class of rules that are more amenable to automatic recovery: rules that are *local*. We define a *local* Tree Association Rule (TAR) as one in which the consequent sub-tree exists within the antecedent sub-tree. This matches the same definition of TARs introduced by Mazuran *et al.* [25]. Based on this definition, we note that local TARs must be intra-directive (scope) and must be child-of (location). Three examples of local TARs (each of which our rule miner is able to discover automatically) are given in Figs. 3(c) and 4(c). In general, the task of finding arbitrary TARs from a corpus of hundreds of thousands of trees is computationally infeasible. By focusing on local TARs, the task of automatic mining becomes tractable.

To identify local TARs binnacle collects, for each node type of interest, the set of all sub-trees with roots of the given type (e.g.,



(a) Stage I: The enforcement engine attempts to match the TAR's antecedent (shown in the outlined box above). A match is found when the subtree in a TAR's antecedent can be aligned with any subtree in the input tree. All three rules given in Fig. 3 have antecedents that match the above tree.

(b) Stage II: If the enforcement engine matches the TAR's antecedent, then, depending on the *location* and *scope* of the TAR, the enforcement engine will *bind* one of the five shaded regions above. For the rule given in Fig. 3(a) (intra-directive preceding), region (2) is matched. For the rule in Fig. 3(b) (intra-directive following), region (5) is matched. The darker shaded regions (1, 4) are the inter-directive variants of regions (2, 5).

(c) Stage III: The enforcement engine searches for the consequent in the bound region. For the rule in Fig. 3(a), the blue shaded region is bound and the consequent (shown with a dashed black outline) is matched; therefore, the rule in Fig. 3(a) has been validated. Conversely, for the rule in Fig. 3(c), the green region is bound and there are no matches for the consequent of this rule (represented by the dashed red box); therefore, the rule in Fig. 3(c) has been violated.

Fig. 6: `binnacle`'s rule engine applied to an example Dockerfile

all sub-trees with APT-GET as the root). On this set of sub-trees, `binnacle` employs frequent sub-tree mining [13] to recover a set of likely consequents. Specifically, binnacle uses the CMTreeMiner algorithm [14] to identify frequent *maximal*, *induced*, *ordered* sub-trees. *Induced* indicates that all "child-of" relationships in the sub-tree exist in the original tree (as opposed to the more permissive "descendent-of" relationship, which defines an *embedded* sub-tree). *Ordered* signifies that order of the child nodes in the sub-tree matters (as opposed to *unordered* sub-trees). A frequent sub-tree is *Maximal* for a given support threshold if there is no super-tree of the sub-tree with occurrence frequency above the support threshold (though there may be sub-trees of the given sub-tree that have a higher occurrence frequency). For more details on frequent sub-trees, see Chi *et al.* [13].

`binnacle` returns rules in which the antecedent is the root node of a sub-tree (where the type of the root node matches the input node-type) and the consequent is a sub-tree identified by the frequent sub-tree miner.

An example of the rule-mining process is given in Fig. 4. In the first stage of rule mining, all sub-trees with the same root node-type (`APT-GET-INSTALL`) are grouped together and collected. For each group of sub-trees with the same root node-type, `binnacle` employs frequent sub-tree mining to find likely consequents. In our example, two frequently occurring sub-trees (in gray and dashed outlines, respectively) are given in Fig. 4(b). Finally, `binnacle` creates local TARs by using the root node as the antecedent and each of the frequent sub-trees as a consequent, as shown in Fig. 4(c). One TAR is created for each identified frequent sub-tree.

3.5 Static Rule Enforcement

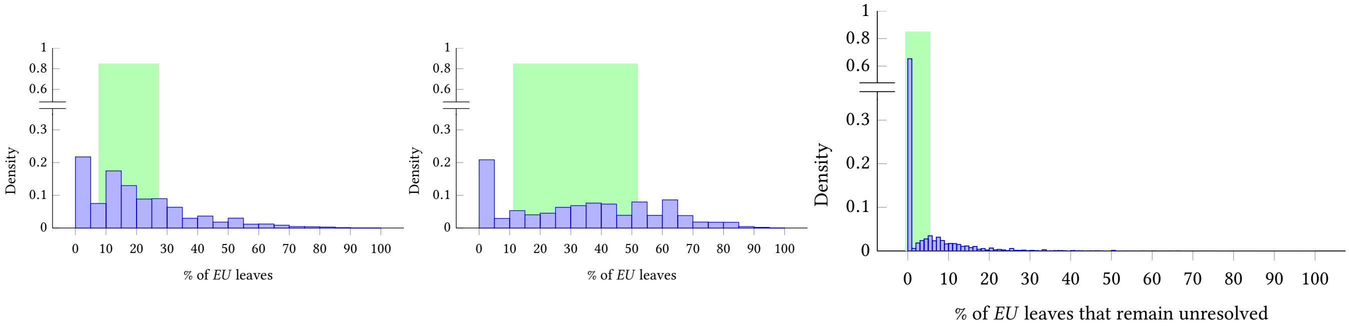
Currently, the state-of-the-art in static Dockerfile support for developers is the VSCode Docker extension [7] and the Hadolint Dockerfile-linting tool [3]. The VSCode extension provides highlighting and basic linting, whereas Hadolint employs a shell parser (ShellCheck [5]—the same shell parser we use) to parse embedded `bash`, similar to our tool's second phase of parsing. The capabilities of these tools represent steps in the right direction but, ultimately, they do not offer enough in the way of deep semantic support. Hadolint does not support parsing of the arguments of individual commands as `binnacle` does in its third phase of parsing. Instead, Hadolint resorts to fuzzy string matching and regular expressions to detect simple rule violations.

`binnacle`'s static rule-enforcement engine takes, as input, a Dockerfile and a set of TARs. `binnacle`'s rule engine runs, for each rule, three stages of processing on the input corpus:

(1) Stage I: The Dockerfile is parsed into a tree representation, and the enforcement engine attempts to match the TAR's antecedent (by aligning it with a sub-tree in the input tree). If no matches are found, the engine continues processing with the next TAR. If a match is found, then the enforcement engine continues to Stage II. This process is depicted in Fig. 6(a).

(2) Stage II: Depending on the *scope* and *location* of the given TAR, the enforcement engine binds a region of the input tree. This region is where, in Stage III, the enforcement engine will look for a sub-tree with which the consequent can be aligned. Fig. 6(b) depicts this process, and highlights the various possible binding regions in the example input tree.

(3) Stage III: Given a TAR with a matched antecedent and a bound region of the input tree, the enforcement engine attempts to align the consequent of the TAR with a sub-tree within the bound region. If the engine is able to find such an alignment, then the rule has been *satisfied*. If not, the rule has been *violated*. Fig. 6(c) depicts this process and both possible



(a) Density histogram of M1 (the fraction of leaves that are *EU* after the first phase of parsing). On average, 19.3% of leaves (median 16.7%) are *EU* at this phase.

(b) Density histogram of M2 (the fraction of leaves that are *EU* after the second phase of parsing). On average, 33.2% of leaves (median 33.3%) are *EU* at this phase.

(c) Density histogram of M3 (the fraction of leaves that were *EU* after the second phase of parsing and unresolved in the third phase). On average, just 3.7% of leaves (median 0.0%) remained *EU*.

Fig. 7: Density histograms showing the distributions of our three metrics (M1,M2, and M3). The green shaded box in each plot highlights the interquartile range for each distribution (the middle 50%).

outcomes: for the rule in Fig. 3(a), the matched antecedent is shown with a thick black outline, the bound region is shown in blue, and the matched consequent is shown with a dashed black outline. In contrast, for the rule in Fig. 3(c), the matched antecedent is the same as above, the bound region is shown in green; however, the tree is missing the consequent, represented by the dashed red sub-tree.

The implementation of binnacle's enforcement engine utilizes a simple declarative encoding for the TARs. To reduce the bias in the manually extracted *Gold Rules* (introduced in §3.2), we used binnacle's static rule-enforcement engine and the Gold Set of Dockerfiles (introduced in §2) to gather statistics that we used to filter the *Gold Rules*. For each of the 23 rules (encoded as Tree Association Rules), we made the following measurements: (i) the *support* of the rule, which is the number of times the rule's antecedent is matched, (ii) the *confidence* of the rule, which is the percentage of occurrences of the rule's consequent that match successfully, given that the rule's antecedent matched successfully, and (iii) the *violation rate* of the rule, which is the percentage of occurrences of the antecedent where the consequent is not matched. Note that our definitions of *support* and *confidence* are the same as that used in traditional association rule mining [10]. We validated our *Gold Rules* by keeping only those rules with *support* greater than or equal to 50 and *confidence* greater than or equal to 75% on the *Gold Set*. These support and confidence measurements are given in Table 1. By doing this filtering, we increase the selectivity of our *Gold Rules* set, and reduce the bias of our manual selection process. Of the original 23 rules in our *Gold Rules*, 16 pass the minimum-support threshold and, of those 16 rules, 15 pass the minimum-confidence threshold. Henceforth, we use the term *Gold Rules* to refer to the 15 rules that passed quantitative filtering. These 15 rules are highlighted, in gray, in Table 1.

Together, binnacle's phased parser, rule miner, and static rule-enforcement engine enable both rule learning and the enforcement of learned rules. Fig. 1 depicts how these tools interact to provide the aforementioned features. Taken together, the binnacle toolset fills the need for structure-aware analysis of DevOps artifacts and provides a foundation for continued research into improving the state-of-the-art in learning from, understanding, and analyzing DevOps artifacts.

4 EVALUATION

In this section, for each of the three core components of the binnacle toolset's learning and enforcement tools, we measure and analyze quantitative results relating to the efficacy of the techniques behind these tools. All experiments were performed on a 12-core workstation (with 32GB of RAM) running Windows 10 and a recent version of Docker.

4.1 Results: Phased Parsing

To understand the impacts of phased parsing, we need a metric for quantifying the amount of *useful information* present in our DevOps artifacts (represented as trees) after each stage of parsing. The metric we use is the fraction of leaves in our trees that are *effectively uninterpretable (EU)*. We define a leaf as *effectively uninterpretable (EU)* if it is, after the current stage of parsing, a string literal that could be further refined by parsing the string with respect to the grammar of an additional embedded language. (We will also count nodes explicitly marked as unknown by our parser as being *EU*.) For example, after the first phase of parsing (the top-level parse), a Dockerfile will have nodes in its parse tree that represent embedded bash—these nodes are *EU* at this stage because they have further structure that can be discovered given a bash parser; however, after the first stage of parsing, these leaves are simply treated as literal values, and therefore marked *EU*.

We took three measurements over the corpus of 178,000 unique Dockerfiles introduced in §2: (M1) the distribution of the fraction of leaves that are *EU* after the first phase of parsing, (M2) the distribution of the fraction of leaves that are *EU* after the second phase of parsing, and (M3) the distribution of the fraction of leaves that are *EU* after the second phase of parsing and unresolved during the third phase of parsing.[4]

[4] For (M3) we make a relative measurement: the reason for using a different metric is to accommodate the large number of new leaf nodes that the third phase of parsing

$$\frac{\textsc{Child-Of}\quad \texttt{(APK-ADD [*])}}{\texttt{(FLAG-NO-CACHE)}}$$

(a) A Gold rule

$$\frac{\textsc{Child-Of}\quad \texttt{(SC-CURL-URL [*])}}{\texttt{(ABS-URL-PROTOCOL-HTTPS)}}$$

(b) A Semantic rule

$$\frac{\textsc{Child-Of}\quad \texttt{(CP [*])}}{\texttt{(CP-PATH) (CP-PATH)}}$$

(c) A Syntactic rule

$$\frac{\textsc{Child-Of}\quad \texttt{(SED [*])}}{\texttt{(FLAG-IN-PLACE)}}$$

(d) An Ungeneralizable rule

Fig. 8: Four examples of actual rules recovered by binnacle's automated miner. Through abstraction, interesting semantic rules, such as using HTTPS URLs with curl, are captured.

Density histograms that depict the three distributions are given in Fig. 7. As shown in Fig. 7, after the first phase of parsing, the trees in our corpus have, on average, 19.3% *EU* leaves. This number quantifies the difficulty of reasoning over DevOps artifacts without more sophisticated parsing. Furthermore, the nodes in the tree most likely to play a role in rules happen to be the *EU* nodes at this stage. (This aspect is something that our quantitative metric does not take into account and hence over-estimates the utility of the representation available after Phase I and Phase II.)

Counterintuitively, the second phase of parsing makes the situation worse: on average, 33.2% of leaves in second-phase trees are *EU*. Competing tools, like Hadolint, work over DevOps artifacts with a similar representation. In practice, competing tools must either stay at what we consider a Phase I representation (just a top-level parse) or utilize something similar to our Phase II representations. Such tools are faced with the high fraction of *EU* leaves present in a Phase II AST. Tools using Phase II representations, like Hadolint, are forced to employ regular expressions or other fuzzy matching techniques as part of their analysis.

Finally, we use our parser generator and the generated parsers for the top-50 commands to perform a third phase of parsing. The plot in Fig. 7(c) shows the M3 distribution obtained after performing the third parsing phase on our corpus of Dockerfiles. At this stage, almost all of the *EU* nodes are gone—on average, only 3.7% of leaves that were *EU* at Phase II remain *EU* in Phase III. In fact, over 65% of trees in Phase II had all *EU* leaves resolved after the third phase of parsing. These results provide concrete evidence of the efficacy of our phased-parsing technique, and, in contrast to what is possible with existing tools, the Phase III structured representations are easily amenable to static analysis and rule mining.

4.2 Results: Rule Mining

We applied `binnacle`'s rule miner to the Gold Set of Dockerfiles defined in §2. We chose the Gold Set as our corpus for rule learning because it presumably contains Dockerfiles of high quality. As described in §3.4, `binnacle`'s rule miner takes, as input, a corpus of trees and a set of node types. We chose to mine for patterns using any new node type introduced by the third phase of parsing. We selected these node types because (i) they represent new information gained in the third phase of our phased-parsing process, and (ii) all rules in our manually collected *Gold Rules* set used nodes created in this phase. Rules involving these new nodes (which come from the most deeply nested languages in our artifacts) were invisible to prior work.

To evaluate `binnacle`'s rule miner, we used the *Gold Rules* (introduced in §3.2). From the original 23 *Gold Rules* we removed 8 rules that did not pass a set of quantitative filters—this filtering is described more in §3.5. Of the remaining 15 *Gold Rules*, there are 9 rules that are *local* (as defined in §3.4). In principal, these 9 rules are all extractable by our rule miner. Furthermore, it is conceivable that there exist interesting and useful rules, outside of the *Gold Rules*, that did not appear in the dockerfile changes that we examined in our manual extraction process. To assess `binnacle`'s rule miner we asked the following three questions:

- (Q1) How many rules are we able to extract from the data automatically?
- (Q2) How many of these rules match one of the 9 local *Gold Rules*? (Equivalently, what is our *recall* on the set of local *Gold Rules*?)
- (Q3) How many new rules do we find, and, if we find new rules (outside of our local *Gold Rules*), what can we say about them (e.g., are the new rules useful, correct, general, etc.)?

For (Q1), we found that `binnacle`'s automated rule miner returns a total of 26 rules. `binnacle`'s automated rule miner is selective enough to produce a small number of output rules—this selectivity has the benefit of allowing for easy manual review.

To provide a point of comparison, we also ran a traditional association rule miner over sequences of tokens in our Phase III ASTs (we generated these sequences via a pre-order traversal). The association rule miner returned thousands of possible association rules. The number of rules could be reduced, by setting very high confidence thresholds, but in doing so, interesting rules could be missed.

For (Q2), we found that two thirds (6 of 9) local *Gold Rules* were recovered by `binnacle`'s rule miner. Because `binnancle`'s rule miner is based on frequent sub-tree mining, it is only capable of returning rules that, when checked against the corpus they were mined from, have a minimum confidence equal to the minimum support supplied to the frequent sub-tree miner.

In addition to measuring recall on the local *Gold Rules*, we also examined the rules encoded in Hadolint to identify all of its rules that were local. Because Hadolint has a weaker representation of Dockerfiles, we are not able to translate many of its rules into local TARs. However, there were three rules that fit the definition of local TARs. Furthermore, `binnacle`'s automated miner was able to recover each of those three rules (one rule requires the use of `apt-get install`'s `-y` flag, another requires the use of `apt-get install`'s `–no-install-recommends` flag, and the third requires the use of `apk add`'s `–no-cache` flag).

To classify the rules returned by our automated miner, we assigned one of the following four classifications to each of the 26 rules returned:

introduces. Without this adjustment, one could argue that our measurements are biased because the absolute fraction of *EU* leaves would be low due to the sheer number of new leaves introduced by the third parsing phase. To avoid this bias, we measure the fraction of *previously EU* leaves that remain unresolved, as opposed to the absolute fraction of *EU* leaves that remain after the third phase of parsing (which is quite small due to the large number of new leaves introduced in the third phase).

- Syntactic: these are rules that enforce simple properties—for example, a rule encoding the fact that the cp command takes two paths as arguments (see Fig. 8(c)).
- Semantic: these are rules that encode more than just syntax. For example, a rule that says the URL passed to the curl utility must include the https:// prefix (see Fig. 8(b)).
- Gold: these are rules that match, or supersede, one of the rules in our collection of *Gold Rules* (see Fig. 8(a)).
- Ungeneralizable: these are rules that are correct on the corpus from which they were mined, but, upon further inspection, seem unlikely to generalize. For example, a rule that asserts that the sed utility is always used with the -in-place flag is ungeneralizable (see Fig. 8(d)).

To answer (Q3), we assigned one of the above classifications to each of the automatically mined rules. We found that, out of 26 rules, 12 were syntactic, 4 were semantic, 6 were gold, and 4 were ungeneralizable. Fig. 8 depicts a rule that was mined automatically in each of the four classes. Surprisingly, binnacle's automated miner discovered 16 new rules (12 syntactic, 4 semantic) that we missed in our manual extraction. Of the newly discovered rules, one could argue that only the semantic rules are interesting (and, therefore, one might expect a human to implicitly filter out syntactic rules during manual mining). We would argue that even these syntactic rules are of value. The lack of basic validation in tools like VS Code's Docker extension creates a use case for these kind of basic structural constraints. Furthermore, the 4 novel semantic rules include things such as: (i) the use of the -L flag with curl, following redirects, which introduces resilience to resources that may have moved, (ii) the use of the -p flag with mkdir, which creates nested directories when required, and (iii) the common practice of preferring soft links over hard links by using ln's -s flag. With (Q3), we have demonstrated the feasibility of automated mining for Dockerfiles—we hope that these efforts inspire further work into mining from Dockerfiles and DevOps artifacts in general.

4.3 Results: Rule Enforcement

Using the 15 *Gold Rules*, we measured the *average violation rate* of the *Gold Rules* with respect to the Gold Dockerfiles (§2). The *average violation rate* is the arithmetic mean of the violation rates of each of the 15 *Gold Rules* with respect to the Gold Dockerfiles. This measurement serves as a kind of baseline—it gives us a sense of how "good" Dockerfiles written by experts are with respect to the *Gold Rules*. The average violation rate we measured was 6.65%, which, unsurprisingly, is quite low. We also measured the average violation rate of the *Gold Rules* with respect to our overall corpus. We hypothesized that Dockerfiles "in the wild" would fare worse, with respect to violations, than those written by experts. This hypothesis was supported by our findings: the average violation rate was 33.15%. We had expected an increase in the violation rate, but were surprised by the magnitude of the increase. These results highlight the dire state of static DevOps support: Dockerfiles authored by non-experts are nearly *five times* worse when compared to those authored by experts. Bridging this gap is one of the overarching goals of the binnacle ecosystem.

We also obtained a set of approximately 5,000 Dockerfiles from the source-code repositories of an industrial source, and assessed their quality by checking them against our *Gold Rules*. To our surprise, the violation rate was no lower for these industrial Dockerfiles. This result provides evidence that the quality of Dockerfiles suffers in industry as well, and that there is a need for tools such as binnacle to aid industrial developers.

5 RELATED WORK

Our paper is most closely related to the work of Sidhu *et al.* [30], who explored reuse in CI specifications in the specific context of Travis CI, and concluded that there was not enough reuse to develop a "tool that provides suggestions to build CI specifications based on popular sequences of phases and commands." We differ in the DevOps artifact targeted (Dockerfiles versus Travis CI files), representation of the configuration file, and the rule-mining approach.

In a related piece of work, Gallaba and McIntosh [17] analyzed the use of Travis CI across nearly 10,000 repositories in GitHub, and identified best practices based on documentation, linting tools, blog posts, and stack-overflow questions. They used their list of best practices to deduce four anti-patterns, and developed Hansel, a tool to identify anti-patterns in Travis CI config files, and Gretel, a tool to automatically correct them. Similar to our second phase of parsing, they used a bash parser (BashLex) to gain a partial understanding of the shell code in config files.

Zhang *et al.* [39] examined the impact of changes to Dockerfiles on build time and quality issues (via the Docker linting tool Hadolint). They found that fewer and larger Docker layers results in lower latency and fewer quality issues in general, and that the architecture and trajectory of Docker files (how the size of the file changes over time) impact both latency and quality. Many of the rules in our Gold Set, and those learned by binnacle, would result in lower latency and smaller images if the rules were followed.

Xu *et al.* [34] described a specific kind of problem in Docker image creation that they call the "Temporary File Smell." Temporary files are often created but not deleted in Docker images. They present two approaches for identifying such temporary files. In this paper, we also observed that removing temporary files is a best-practice employed by Dockerfile experts and both our manual Gold Set and our learned rules contained rules that address this.

Zhang *et al.* [38] explored the different methods of continuous deployment (CD) that use containerized deployment. While they found that developers see many benefits when using CD, adopting CD also poses many challenges. One common way of addressing them is through containerization, typically using Docker. Their findings also reinforce the need for developer assistance for DevOps: they concluded that "Bad experiences or frustration with a specific CI tool can turn developers away from CI as a practice."

Our work falls under broader umbrella of "infrastructure-as-code". This area has received increasing attention recently [28]. As examples, Sharma *et al.* examined quality issues, so-called *smells*, in software-configuration files [29], and Jiang *et al.* examined the coupling between infrastructure-as-code files and "traditional" source-code files.

There have been a number of studies that mine Docker artifacts as we do. Xu and Marinov [35] mined container-image repositories such as DockerHub, and discussed the challenges and opportunities that arise from such mining. Zerouali *et al.* [37] studied vulnerabilities in Docker images based on the versions of packages installed in them. Guidotti *et al.* [18] attempted to use Docker-image metadata to determine if certain combinations of image attributes led to increased popularity in terms of stars and pulls. Cito *et al.* [15] conducted an empirical study of the Docker ecosystem on GitHub by mining over 70,000 Docker files, and examining how they evolve, the types of quality issues that arise in them, and problems when building them.

A number of tools related to Dockerfiles have been developed in recent years as well.

Brogi *et al.* [12] found that searching for Docker images is currently a difficult problem and limited to simple keyword matching. They developed DockerFinder, a tool that allows multi-attribute search, including attributes such as image size, software distribution, or popularity.

Yin *et al.* [36] posited that tag support in Docker repositories would improve reusability of Docker images by mitigating the discovery problem. They addressed this issue by building STAR, a tool that uses latent dirichlet allocation to automatically recommend tags.

Docker files may need to be updated when the requirements of the build environment or execution environment changes. Hassan *et al.* [19] developed Rudsea, a tool that can recommend updates to Dockerfiles based on analyzing changes in assumptions about the software environment and identifying their impacts.

To tackle the challenge of creating the right execution environment for python code snippets (*e.g.*, from Gists or StackOverflow) Horton and Parnin [21] developed DockerizeMe, a tool which infers python package dependencies and automatically generates a Dockerfile that will build an execution environment for pieces of python code.

6 THREATS TO VALIDITY

We created tools and techniques that are general in their ability to operate over DevOps artifacts with embedded shell, but we focused our evaluation on Dockerfiles. It is possible that our findings do not translate directly to other classes of DevOps artifacts. We ingested a large amount of data for analysis, and, as part of that process, we used very permissive filtering. It is possible that our corpus of Dockerfiles contains files that are not Dockerfiles, duplicates, or other forms of noise. It is also possible that there are bugs in the infrastructure used to collect repositories and Dockerfiles. To mitigate these risks we kept a log of the data we collected, and verified some coarse statistics through other sources (e.g., we used GitHub's API to download data and then cross-checked our on-disk data against GitHub's public infrastructure for web-based search). Through these cross-checks we were able to verify that, for the over 900,000 repositories we ingested, greater than 99% completed the ingestion process successfully. Furthermore, of the approximately 240,000 likely Dockerfiles we identified, 91% (219,000) made it through downloading, parsing, and validation. Of this set of files, approximately 81% (178,000) were unique based on their SHA1 hash. Of the files rejected during processing (downloading, parsing, and validation), most were either malformed Dockerfiles or files with names matching our `.*dockerfile.*` filter that were not actual Dockerfiles (e.g., `docker-compose.yml` files).

We identified a Gold Set of Dockerfiles and used these files as the ideal standard for the Dockerfiles in our larger corpus. It is possible that developers do not want to achieve the same level of quality as the files in our Gold Set. It is also possible that the Gold Set is too small and too specific to be of real value. It is conceivable, but unlikely, that the Gold Set is not representative of good practice. Even if that were the case, our finding still holds that there is a significant difference between certain characteristics of Dockerfiles written by (presumed) Docker experts and those written by garden-variety GitHub users. We acknowledge that the average violation rate of our Gold Rules is only a proxy for quality—but, given the data and tools currently available, it is a reasonable and, crucially, measurable choice of metric. For rule mining, we created, manually, a set of Gold Rules against which we benchmarked our automated mining. Because the results of automated mining did not agree with three of the manually extracted rules, there is evidence that the manual process did have some bias. We sought to mitigate this issue through the use of quantitative filtering; after filtering, we retained only 65% of our original Gold Rules.

7 CONCLUSION

Thus far, we have identified the ecosystem of DevOps tools and artifacts as an ecosystem in need of greater support both academically and industrially. We found that, on average, Dockerfiles on GitHub are nearly *five times worse*, with respect to violations of our *Gold Rules*, compared to Dockerfiles written by experts. Furthermore, we found that industrial Dockerfiles are no better. Through automated rule mining and static rule enforcement, we created tools to help bridge this gap in quality. Without increased developer assistance, the vast disparity between the quality of DevOps artifacts authored by experts and non-experts is likely to continue to grow.

There are a number of pieces of follow-on work that we hope to pursue. We envision the `binnacle` tool, the data we have collected, and the analysis we have done on Dockerfiles as a foundation on which new tools and new analysis can be carried out. To that end, we plan to continue to evolve the `binnacle` ecosystem by expanding to more DevOps artifacts (Travis, CircleCI, etc.). Additionally, the encoding of rules we utilize has the advantage of implicitly encoding a repair (or, at least, part of a repair—localizing the insertion point for the implicit repair may be a challenge). Furthermore, the kinds of rules that we mine are limited to *local* rules. We believe that more rules may be within the reach of automated mining. Finally, we hope to integrate `binnacle`'s mined rules and analysis engine into language servers and IDE plugins to provide an avenue for collecting real feedback that can be used to improve the assistance we provide to DevOps developers.

ACKNOWLEDGMENTS

Supported, in part, by a gift from Rajiv and Ritu Batra and by ONR under grants N00014-17-1-2889 and N00014-19-1-2318. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies.

REFERENCES

[1] 2019. CircleCI - Continuous Integration and Delivery. https://circleci.com [Online; accessed 21. Aug. 2019].

[2] 2019. Enterprise Container Platform | Docker. https://www.docker.com [Online; accessed 21. Aug. 2019].

[3] 2019. hadolint/hadolint. https://github.com/hadolint/hadolint [Online; accessed 21. Aug. 2019].

[4] 2019. Jenkins - Open Source Automation Server. https://jenkins.io [Online; accessed 21. Aug. 2019].

[5] 2019. koalaman/shellcheck. https://github.com/koalaman/shellcheck [Online; accessed 21. Aug. 2019].

[6] 2019. Kubernetes - Production-Grade Container Orchestration. https://kubernetes.io [Online; accessed 21. Aug. 2019].

[7] 2019. microsoft/vscode-docker. https://github.com/microsoft/vscode-docker [Online; accessed 21. Aug. 2019].

[8] 2019. Terraform - Write, Plan, and Create Infrastructure as Code. https://www.terraform.io [Online; accessed 21. Aug. 2019].

[9] 2019. Travis CI - Test and Deploy Your Code with Confidence. https://travis-ci.org [Online; accessed 21. Aug. 2019].

[10] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. 1993. Mining association rules between sets of items in large databases. In *Acm sigmod record*, Vol. 22. ACM, 207–216.

[11] Leonard J. Bass, Ingo M. Weber, and Liming Zhu. 2015. *DevOps - A Software Architect's Perspective.* Addison-Wesley. http://bookshop.pearson.de/devops.html?productid=208463

[12] A. Brogi, D. Neri, and J. Soldani. 2017. DockerFinder: Multi-attribute Search of Docker Images. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*. 273–278. https://doi.org/10.1109/IC2E.2017.41

[13] Yun Chi, Richard R Muntz, Siegfried Nijssen, and Joost N Kok. 2005. Frequent subtree mining–an overview. *Fundamenta Informaticae* 66, 1-2 (2005), 161–198.

[14] Yun Chi, Yi Xia, Yirong Yang, and Richard R Muntz. 2005. Mining closed and maximal frequent subtrees from databases of labeled rooted trees. *IEEE Transactions on Knowledge and Data Engineering* 17, 2 (2005), 190–202.

[15] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall. 2017. An Empirical Analysis of the Docker Container Ecosystem on GitHub. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 323–333. https://doi.org/10.1109/MSR.2017.67

[16] Jennifer Davis and Ryn Daniels. 2016. *Effective DevOps: building a culture of collaboration, affinity, and tooling at scale.* O'Reilly Media, Inc.

[17] K. Gallaba and S. McIntosh. 2018. Use and Misuse of Continuous Integration Features: An Empirical Study of Projects that (mis)use Travis CI. *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2838131

[18] Riccardo Guidotti, Jacopo Soldani, Davide Neri, and Antonio Brogi. 2018. Explaining Successful Docker Images Using Pattern Mining Analysis. In *Software Technologies: Applications and Foundations*, Manuel Mazzara, Iulian Ober, and Gwen Salaün (Eds.). Springer International Publishing, Cham, 98–113.

[19] Foyzul Hassan, Rodney Rodriguez, and Xiaoyin Wang. 2018. RUDSEA: Recommending Updates of Dockerfiles via Software Environment Analysis. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, New York, NY, USA, 796–801. https://doi.org/10.1145/3238147.3240470

[20] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 426–437.

[21] Eric Horton and Chris Parnin. 2019. DockerizeMe: Automatic Inference of Environment Dependencies for Python Code Snippets. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 328–338. https://doi.org/10.1109/ICSE.2019.00047

[22] Gene Kim, Jez Humble, Patrick Debois, and John Willis. 2016. *The DevOps Handbook:: How to Create World-Class Agility, Reliability, and Security in Technology Organizations.* IT Revolution.

[23] Lucy Ellen Lwakatare, Pasi Kuvaja, and Markku Oivo. 2015. Dimensions of devops. In *International conference on agile software development*. Springer, 212–217.

[24] Visual Studio Code Marketplace. 2020. Docker. https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-docker [Online; accessed 29. Jan. 2020].

[25] Mirjana Mazuran, Elisa Quintarelli, and Letizia Tanca. 2009. Mining tree-based association rules from XML documents.. In *Proceedings of the Seventeenth Italian Symposium on Advanced Database Systems*. 109–116.

[26] Shaun Phillips, Thomas Zimmermann, and Christian Bird. 2014. Understanding and improving software build teams. In *Proceedings of the 36th international conference on software engineering*. ACM, 735–744.

[27] Portworx. 2017. Annual Container Adoption Report. https://portworx.com/2017-container-adoption-survey/ [Online; accessed 21. Aug. 2019].

[28] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie Williams. 2019. A systematic mapping study of infrastructure as code research. *Information & Software Technology* 108 (2019), 65–77. https://doi.org/10.1016/j.infsof.2018.12.004

[29] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does your configuration code smell?. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 189–200.

[30] Puneet Kaur Sidhu, Gunter Mussbacher, and Shane McIntosh. 2019. Reuse (or Lack Thereof) in Travis CI Specifications: An Empirical Study of CI Phases and Commands. In *Proc. of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 524–533.

[31] Daniel Ståhl and Jan Bosch. 2016. Industry application of continuous integration modeling: a multiple-case study. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 270–279.

[32] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 805–816.

[33] David Gray Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. 2019. A Conceptual Replication of Continuous Integration Pain Points in the Context of Travis CI. (2019).

[34] J. Xu, Y. Wu, Z. Lu, and T. Wang. 2019. Dockerfile TF Smell Detection Based on Dynamic and Static Analysis Methods. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. 185–190. https://doi.org/10.1109/COMPSAC.2019.00033

[35] T. Xu and D. Marinov. 2018. Mining Container Image Repositories for Software Configuration and Beyond. In *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*. 49–52.

[36] K. Yin, W. Chen, J. Zhou, G. Wu, and J. Wei. 2018. STAR: A Specialized Tagging Approach for Docker Repositories. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. 426–435. https://doi.org/10.1109/APSEC.2018.00057

[37] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona. 2019. On the Relation between Outdated Docker Containers, Severity Vulnerabilities, and Bugs. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 491–501. https://doi.org/10.1109/SANER.2019.8668013

[38] Yang Zhang, Bogdan Vasilescu, Huaimin Wang, and Vladimir Filkov. 2018. One Size Does Not Fit All: An Empirical Study of Containerized Continuous Deployment Workflows. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 295–306. https://doi.org/10.1145/3236024.3236033

[39] Yang Zhang, Gang Yin, Tao Wang, Yue Yu, and Huaimin Wang. 2018. An insight into the impact of dockerfile evolutionary trajectories on quality and latency. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 138–143.

[40] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. The impact of continuous integration on other software development practices: a large-scale empirical study. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 60–71.