Analysing Software Supply Chains of Infrastructure as Code: Extraction of Ansible Plugin Dependencies

Ruben Opdebeeck
Ruben.Denzel.Opdebeeck@vub.be
Vrije Universiteit Brussel, Belgium

Bram Adams
bram.adams@queensu.ca
Queen's University, Canada

Coen De Roover
Coen.De.Roover@vub.be
Vrije Universiteit Brussel, Belgium

***Abstract*—The digital infrastructures supporting modern software have grown too complex to manage by hand. Therefore, Infrastructure as Code (IaC) has become a widely adopted practice to programmatically automate deploying such infrastructures. As infrastructure code may rely on third-party libraries and packages, understanding the software supply chains generated by these deployment dependencies is crucial to ensure reproducibility and security of software deployments. Nonetheless, *deployment software supply chains* remain an understudied topic. This paper aims to bridge this gap by first investigating which types of third-party software IaC may depend upon, then building an automated mechanism to identify such dependencies from infrastructure implementations. We focus our investigation on Ansible, one of the most popular IaC tools, and its plugins, which implement the interactions with the deployment platforms under configuration. From a manual analysis of 266 documented third-party requirements of Ansible plugins, we construct a taxonomy of 7 types of third-party software dependencies and their properties. We also found that a plugin's dependencies are typically only described informally in the plugin's documentation, which may be unstructured, incorrect, or incomplete, which encumbers the automatic generation of Software Bills of Materials (SBOMs) for deployment code. Therefore, we design an automated Software Composition Analysis (SCA) that extracts these dependencies from an Ansible plugin's implementation, leveraging 5 dependency implementation patterns identified in our manual analysis. This approach achieves a recall of 61%–77% and a precision of 74%–95%. Finally, we apply the SCA in a large-scale quantitative experiment on 11,241 plugins, and find that 38% have third-party dependencies. The taxonomy presented in this paper can serve as a reference to design deployment SBOMs for these plugins, whereas our SCA forms a first step towards automatically generating such SBOMs.**

***Index Terms*—software supply chain, infrastructure as code, deployment automation, third-party dependencies, software composition analysis**

I. Introduction

Automating software deployments to increase their reliability and reproducibility is a key activity in the DevOps approach. To this end, *Infrastructure as Code* (IaC) [1] has become an established practice [2], enabling developers to specify infrastructure automation as executable source code. Consequently, tools that enable IaC form important parts of *software supply chains*, i.e., the components, tools, and processes used to build a software product. Analysing software supply chains is vital, especially from a security perspective, as attacks on software supply chains have become increasingly common [3]. Their alarmingly high risk has led to new government legislations in the US [4] and the EU [5] aimed at strengthening them by, among others, adopting Software Bills of Materials (SBOMs)—structured, machine-readable specifications of the composition of software products that aid in identifying vulnerable or outdated dependencies [6].

Aside from constituting parts of software supply chains, infrastructure automation code also has its own software supply chain, which we coin the *deployment software supply chain*. These may comprise IaC artefacts from within the IaC tool's own ecosystem, but also software from several general-purpose software ecosystems. For instance, to provision a cloud machine, the infrastructure code may depend on OS packages (e.g., to create cryptographic keys), the remote API of the cloud service (e.g., Amazon's AWS APIs), and development libraries (e.g., a language-specific wrapper around the remote API). Similar to traditional software supply chains, the identification of vulnerable dependencies, licence violations, etc. requires analysing these deployment software supply chains and generating SBOMs for IaC. However, to the best of our knowledge, an automated analysis of deployment software supply chains does not yet exist.

We aim to bridge this gap by designing and empirically evaluating such an automated analysis for Ansible, one of the most popular IaC tools today [7]. Specifically, we investigate two fundamental parts that are necessary for an automated analysis. First, we aim to identify the information that needs to be contained in a deployment SBOM for Ansible. Second, we investigate how the dependencies of Ansible infrastructure code can be extracted automatically, forming a stepping stone to automated generation of deployment SBOMs for Ansible.

We focus our investigation on Ansible plugins, which implement the interactions with the configured systems, rather than the Ansible infrastructure specifications themselves. While both may have third-party dependencies, the latter are managed through well-defined machine-readable requirements files [8], which lend themselves well to automated analysis. Contrarily, plugin dependencies, which are the focus of this work, are neither well-defined nor well-structured, being specified only in informal documentation[1], which may be incorrect or incomplete. This encumbers automated analysis, as it is unknown what types of dependencies are to be considered, or how they can be extracted.

[1]https://github.com/ansible/ansible/issues/62733

Therefore, in this paper, we first conduct a qualitative empirical analysis of documented plugin dependencies to identify the types of third-party libraries and packages depended upon by plugins. This provides insights into the information that needs to be contained within a deployment SBOM for Ansible. Afterwards, we qualitatively study plugin implementations to identify third-party dependency management patterns, which we use to design an automated Software Composition Analysis (SCA) that extracts the dependencies from Ansible plugin implementations. This SCA can be built upon to automatically generate deployment SBOMs for Ansible infrastructure code. Finally, we show how this SCA enables analysing Ansible deployment software supply chains at scale by conducting a large-scale quantitative experiment into the prevalence of plugin dependencies.

In summary, this paper presents the following contributions:

- We conduct the first empirical study on deployment dependencies in Ansible Infrastructure as Code applications.
- A taxonomy of 7 types of Ansible plugin dependencies and their properties, providing insights into the information that should be contained within deployment SBOMs for Ansible.
- A catalogue of 5 dependency management practices implemented in Ansible plugins, which can serve as a reference to plugin maintainers to manage their dependencies.
- An automated Software Composition Analysis for Ansible that can be built upon to generate SBOMs for plugins.
- A large-scale quantitative analysis of Ansible dependencies that shows how the SCA enables automated analysis of deployment software supply chains at scale.

A replication package containing our dataset, analysis scripts, and a prototypical implementation of the SCA is available at https://doi.org/10.6084/m9.figshare.27195810.

II. Background

Ansible is a popular automation tool that can be used to apply Infrastructure as Code. Ansible projects consist of an *application layer*, in which practitioners use the Ansible domain-specific language to specify the desired state of infrastructure machines, and a *runtime layer* that interprets these specifications and configures the targeted systems [9].

In the application layer, practitioners write *playbooks*, specifying the tasks required to provision and configure an infrastructure of machines. A playbook executes on the *controller*, a machine controlling the configuration process, which executes each task on the *host* machines to be configured. Figure 1 depicts an example that configures a list of servers (line 1) by deploying a Docker container (lines 3–9).

In the runtime layer, a plugin system allows extending Ansible's functionality. For instance, *filter plugins* and *test plugins* extend Ansible's embedded Jinja2 expression language with new operations to transform data and test predicates respectively. *Inventory plugins* construct an *inventory*, a list of hosts to be configured. *Modules* are special types of plugins that implement the logic behind the steps executed by a task, e.g., `community.docker.docker_container` in Figure 1. Modules differ from other plugins as they are executed in a separate process on each host, rather than on the controller itself. Plugins have to be implemented in Python, while modules can be implemented in any language.

```
- hosts: servers                                    1
  tasks:                                            2
    - name: Deploy Docker container                 3
      community.docker.docker_container:            4
        name: "nginx-server"                        5
        image: "nginx"                              6
        state: "started"                            7
        ports:                                      8
          - "8080:80"                               9
```

Fig. 1: Example of an Ansible playbook.

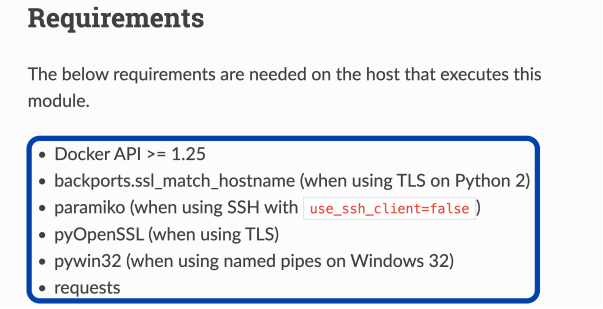

Fig. 2: Excerpt of the `docker_container` documentation.

Earlier versions of Ansible curated plugins in a centralised mono-repository. As the number of plugins grew, Ansible introduced *collections* of plugins and migrated the majority of plugins to dedicated collection repositories. Collections can also be maintained by third parties, and are indexed by Galaxy[2], Ansible's content registry. A collection can be identified uniquely by the combination of its namespace (author) and name, usually separated by a period. For instance, the *community.docker* collection comprises plugins for working with Docker, such as the `docker_container` module in Figure 1, and the *amazon.aws* collection comprises plugins that interact with Amazon's AWS cloud.

A collection may depend on other collections, which can be specified in the collection's metadata manifest. A collection's plugins may also depend on other types of software, such as Python libraries or OS packages. For instance, the `docker_container` module in Figure 1 depends on the Docker API and several Python packages, as depicted in Figure 2. Consequently, the deployment software supply chain of Ansible playbooks comprises various types of third-party software from diverse sources. However, these dependencies can only be specified informally in the plugin's documentation[3], which may be highly unstructured, as exemplified in Figure 2. Moreover, as module plugins execute on remote hosts, these dependencies are to be installed on that remote host, rather than with the collection on the controller. Not

[2]https://galaxy.ansible.com/

[3]Although some collections provide a Python `requirements.txt` file, a preliminary analysis uncovered that these are too coarse-grained to identify plugin dependencies, as not every plugin in a collection uses every dependency.

installing these dependencies will cause the execution of the Ansible playbook using the plugins to fail partway, leaving the remote hosts in a partially configured state.

III. Empirical Study Design

In this section, we describe the design of our empirical study. We investigate the following research questions:

- *RQ_1: Which types of third-party software do Ansible plugins depend on?* We first perform a manual qualitative analysis of the types and characteristics of third-party software dependencies of Ansible plugins, to gain a better understanding of what needs to be represented in deployment SBOMs for Ansible.
- *RQ_2: Do plugin implementations exhibit patterns that indicate dependencies?* The unstructured and informal nature of the plugin documentation encumbers automatically extracting third-party dependencies. However, Ansible recommends plugin maintainers to manually check whether dependencies are satisfied before using them [10]. We therefore qualitatively investigate the plugin implementations to distil implementation patterns that can identify dependencies.
- *RQ_3: To what extent can Ansible plugin dependencies be identified automatically?* We develop a Software Composition Analysis (SCA) based on the dependency management patterns from RQ_2, and evaluate whether it can accurately identify the plugin's dependencies.
- *RQ_4: How prevalent are Ansible plugin dependencies in the wild?* Using the SCA resulting from RQ_3, we conduct a large-scale quantitative study of Ansible plugin dependencies, from which we aim to gain a better understanding of the extent of deployment software supply chains.

A. Data collection

To collect a representative dataset of Ansible plugin collections, we scrape the Ansible Galaxy registry (cf. Section II). For each collection, we use the registry's API to gather information such as its GitHub repository, its releases, and the number of times it has been downloaded from the registry. Then, we augment this data with information from the collection's GitHub repository (if specified), including the number of commits, issue count, star count, etc. We collected the data on 2024–01–26, discovering 2817 collections and 1867 GitHub repositories. We could not collect the information of 175 of these GitHub repositories, e.g., because the repository was removed or set to private.

As our goal is to study unique, open-source Ansible collections, we exclude 1125 of the 2817 collections because they do not have a public GitHub repository. Furthermore, to avoid considering duplicates, when two collections specify the same GitHub repository, we only retain the one with the highest download count. This eliminates another 242 collections.

Second, we apply filtering to ensure that our dataset contains only mature and actively-maintained collections. We exclude 8 collections that are marked as deprecated on Ansible Galaxy. We further exclude another 584 whose repository has not been committed to in the year leading up to the data collection date, and are thus no longer actively maintained. To retain only those that are mature, we exclude 122 collections with fewer than 10 commits. Similarly, we exclude another 165 that have been active for less than half a year, measured as the time between the date of the first and last commit to the repository.

Finally, we want to focus on collections that are widely used in practice. Therefore, we retain only the top 10% most downloaded collections according to Ansible Galaxy statistics. This excludes another 384 collections that have been downloaded less than 15,700 times.

After filtering, the resulting dataset counts 187 widely-used, mature, open-source Ansible collections, representing 92.3% of all downloads in the Ansible Galaxy ecosystem. Table I depicts their summary statistics. For the empirical analysis, we download the latest version of each collection. We also add the *ansible.builtin* collection to the dataset. It is shipped with Ansible and therefore should be included, but is missed by our data collection strategy since it is not distributed through Ansible Galaxy. Our final dataset contains 188 collections.

TABLE I: Dataset statistics, excl. *ansible.builtin*.

Property	Minimum	Median	Mean	Maximum
# downloads	15.7K	136.3K	2.11M	62.5M
# commits	10	269	767	33.7K
Last activity	0 days	18 days	53 days	362 days
Active time	233 days	1370 days	1355 days	3984 days

B. Parse collection documentation

We proceed to extract the requirements specified in the embedded documentation of Ansible collection plugins. The documentation for a plugin is embedded into its Python implementation as a YAML document. We create a script that, given a collection, first enumerates the collection's plugins, then extracts documented requirement lists for each plugin individually. For instance, for the plugin whose documentation is depicted in Figure 2, the script would produce a list of 6 items, each containing the text for one bullet of the highlighted requirements list. For both steps, the script leverages Ansible's `ansible-doc` command, which processes the embedded documentation. We run the script on the latest version of the 188 collections in the dataset. We enumerate 13,721 plugins and successfully parse the documentation of 13,164 plugins. In total, we identify 10,960 requirements belonging to 5537 plugins, forming the population for our study.

C. Open coding of collection dependencies

We manually investigate a statistically significant sample of collection requirements to identify the types of third-party software on which plugins depend (RQ_1), as well as dependency management patterns in the plugin implementations (RQ_2). For the latter, we intend to focus on how plugins check whether dependencies are satisfied, and on what steps plugins take if a dependency is not satisfied.

We first deduplicate documented requirements across different plugins in the same collection, because these plugins may use common implementations, which would introduce bias in our results. This results in 866 unique combinations of

a collection and an individual documented plugin requirement. We then take a random sample of this deduplicated population, using a sample size obtained using Cochran's formula for categorical data, adjusted for small population sizes [11] with 95% confidence, 5% margin of error, and an estimated population proportion of 50% to maximise the sample size. This results in a sample of 266 requirements spanning 51 collections. Then, the first author applies open coding to qualitatively answer RQ_1 and RQ_2. To avoid subjective bias which could be caused by the use of a single labeller, we only label purely objective properties for which any ambiguity can be resolved by consulting the online documentation.

For RQ_1, the labeller determines whether the documented requirement represents a valid software dependency (e.g., "This plugin requires the "requests" Python package"), as opposed to requirements that do not contribute to the deployment software supply chain and are thus outside the scope of this study (e.g., "This plugin requires root privileges"). Valid requirements state the type of dependency, such as the source ecosystem, and related properties, such as whether the dependency has a version constraint. When the dependency type is unclear, the labeller refers to online documentation for the specified dependency. In subsequent iterations, they generalise the labels to create categories of dependencies, and derive the general properties exhibited by dependency-specifying documented requirements.

For RQ_2, for each dependency, we randomly select one plugin whose requirement list contains the dependency. The labeller then inspects the entirety of this plugin's implementation for dependency management concerns. They summarise the means through which the plugin checks whether the dependency is satisfied, and the behaviour the plugin exhibits in case the dependency is not satisfied. Both summaries are assigned as distinct codes for the dependency. After coding each dependency, the labeller generalises the codes to a set of high-level properties that characterise the dependency management, and iteratively refines the properties to capture variations in similar codes. The result of this process is a set of commonly-occurring implementation patterns related to plugin dependency management.

D. Automated Software Composition Analysis

After the qualitative analysis in RQ_2, we will have identified code patterns whose presence in a plugin's implementation are indicative of its management of a dependency. As unstructured dependency-specifying requirements cannot be used to reliably extract deployment supply chains, we now use these patterns to design a Software Composition Analysis that automatically extracts managed dependencies from plugin implementations.

To find pattern instances in a plugin implementation, we will use Joern [12], a multi-language static analysis framework. Joern represents projects as code property graphs (CPGs), the nodes of which represent program elements such as statements and expressions. The edges capture properties of and relationships between these elements, including abstract syntax tree containment, calls between functions, control flow, data flow, and type information. Our approach will build a CPG for all plugins in an Ansible collection, and will evaluate Joern queries against the CPG that find instances of dependency management patterns and thereby identify dependencies.

To answer RQ_3, we will measure the precision and recall of our Software Composition Analysis. The ground truth comprises the manually-validated dependency-specifying documented requirements from RQ_1. We run the SCA on the implementation of the plugins from the ground truth, and manually compare its output to the ground truth to identify true positives. When the SCA reports multiple results that belong to the same dependency (e.g., individual binaries from one OS package), we manually merge the reports into one, consulting online documentation where necessary. We do not distinguish between optional and mandatory dependencies, and do not consider version constraints, as we only need to determine whether and which third-party software plugins rely on.

To compute recall, we mark those entries from the ground truth that are missing from the SCA output as false negatives. Note that when a plugin implementation does not check for the presence of its dependencies, the SCA will fail to identify them. Therefore, we calculate an upper and lower bound on recall. The upper bound is defined as the ratio of identified dependencies over the number of dependencies in the ground truth for which the plugin checks whether the dependency is satisfied. The lower bound is the ratio of identified dependencies over the entire size of the ground truth, including unchecked dependencies.

To calculate precision, when an entry in the SCA output is not in the ground truth, we further inspect the plugin documentation and implementation to determine whether it is a true or false positive. This is necessary because some plugin dependencies may not have been documented, and are therefore missing from the ground truth which is constructed from the plugin documentation. Moreover, some dependencies may be on built-ins, such as built-in Python libraries or OS packages. Although such built-in dependencies are likely to be considered as false positives by plugin users and maintainers, they are depended upon by the plugin. Moreover, dependencies that are typically considered built-in may turn out to be unavailable after all, e.g., the Python `ssl` library may be missing if Python is compiled without SSL support.

Therefore, whether built-in dependencies should be considered true positives depends on context. To avoid subjectivity, we again calculate an upper and lower bound on precision. For the lower bound, we assume that *all* built-in dependencies are false positives, which would be appropriate for developer-oriented tooling. For the upper bound, we do not distinguish between built-in and other dependencies, which would be appropriate to extract the entire deployment software supply chain.

E. Quantitative analysis of plugin dependencies

To answer RQ_4, we apply the SCA resulting from RQ_3 to all 188 collections in the dataset, obtaining lists of dependencies for each plugin. We map each code pattern identified in

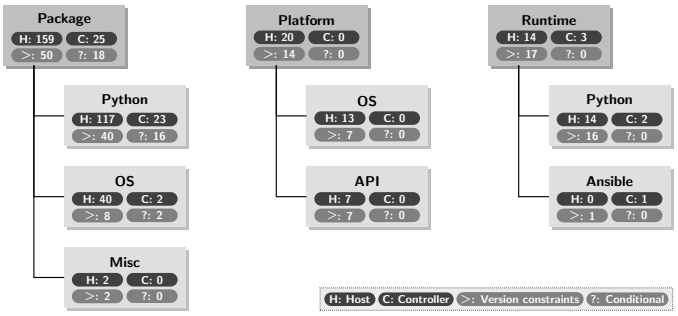

Fig. 3: Taxonomy of dependencies in Ansible collections. Host and controller counts are mutually exclusive.

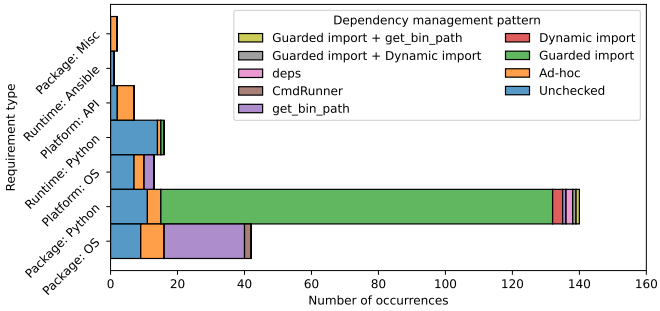

Fig. 4: Frequency of dependency management patterns.

RQ_2 to the most common dependency type the pattern is used for. We use the mappings to automatically identify the types of third-party software in the SCA results. Using the results, we investigate and compare how many plugins depend on the types of third-party software identified in RQ_1.

IV. Qualitative Analysis

In this section, we present the results of our manual qualitative study into the dependencies of and the dependency management implemented by Ansible plugins.

A. RQ_1: Which types of third-party software do Ansible plugins depend on?

Among the sample of 266 requirements from the embedded plugin documentation, we manually identify 221 valid dependency-specifying ones. The other 45 requirements specify generic preconditions, such as user permissions or conditions on the Ansible configuration. We note that the dependency specifications in the plugin documentation are highly unstructured, ranging from merely the dependency name (e.g., "requests") to full sentences (e.g., "Requires `lzma` (standard library of Python 3) or `backports.lzma` (Python 2) if using `xz` format").

In the valid dependencies, we discern 7 distinct types, grouped into 3 categories. The resulting taxonomy is depicted in Figure 3. The most common category is *Package*, comprising third-party software that is typically installed through managers. Python libraries form the majority of this category, followed by operating system packages for package managers like `dpkg` (Debian, Ubuntu, ...) or `rpm` (CentOS, Fedora, ...). The *Misc* subcategory encompasses other package dependencies that are less common, such as Terraform packages. The second category is *Platform*, consisting of operating systems (e.g., macOS) and APIs. The latter refers to interfaces the plugins communicate with, either locally (e.g., the Docker API) or remotely (e.g., remote management systems). The final category consists of *runtime* version dependencies, encompassing minimum versions of Ansible and Python.

We identify 3 orthogonal properties of the 7 categories of collection dependencies. First, we can classify dependencies according to whether they should be installed on the host that is being configured or on the Ansible controller (cf. Section II), which depends on the type of plugin. 86% (190) of the studied dependencies must be satisfied on the remote host machines. We note that host dependencies are more diverse than controller dependencies. Indeed, most controller dependencies are Python packages, while certain types of dependencies, such as those in the *Platform* category, are only needed on the remote host. Second, only 37% of dependency specifications (81) include some form of version constraint, with API and runtime dependencies always specifying a version constraint. These are denoted by $>$ in the taxonomy. Finally, while the majority of dependencies are mandatory, we find 18 (8%) that are either optional or only need to be installed depending on certain conditions, e.g., depending on the arguments given to the plugin. These are denoted by **?** in the taxonomy.

Summary: Plugin documentation does not always specify dependencies, nor in a structured manner. Deployment SBOMs need to discern 7 types of dependencies in 3 categories. Python and OS packages are the most prevalent.

B. RQ_2: Do plugin implementations exhibit patterns that indicate dependencies?

We find that for the majority of dependencies (174 of 221), the plugins check whether the dependency is satisfied. The majority of these checks (67%) happen inside the plugin file itself, while the remaining 33% occur in shared utility files that can be used by multiple plugins which may have different dependency or version requirements.

However, for 47 dependencies (21%), the plugin does not check that it is satisfied, and only 36% of specified version constraints are verified by the plugins. In the *Runtime* category, only 2 Python version requirements are validated by plugins, with the remaining 14 Python versions and 1 Ansible version remaining unchecked. Omitting these checks may lead to run-time crashes with confusing error messages.

1) Dependency management patterns: We identify 5 common implementation patterns for checking whether dependencies are satisfied. Table II summarises them with an example, whereas Figure 4 depicts their frequency per dependency type.

The most common pattern is the *guarded import*, in which developers wrap an import of a Python package in a try-except block. They then assign a variable to indicate whether the import was successful, and check this variable to abort with an error message. Its frequent usage for Python packages is not surprising, as it is recommended in the Ansible documentation [10]. *Dynamic import* is a closely-related but less

TABLE II: Catalogue of dependency management implementation patterns.

Pattern	Use cases	Example	Detection pseudo-query

Guarded import	Python packages, Python version	`HAS_LIB = True`
		`try:`
		`  import lib`
		`except ImportError:`
		`  HAS_LIB = False`
		`if not HAS_LIB:`
		`  <error>`
			`tryStatement`
			`  .where(_.exceptBlock.contains(_.isAssignment))`
			`  .tryBlock.children.filter(_.isImport)`
			`  .map(_.importedName)`

Dynamic import	Python packages	`try:`
		`  importlib.import_module("lib")`
		`except ModuleNotFoundError:`
		`  <error>`
			`callTo("import_module")`
			`  .arguments(0)`
			`  .resolveString()`

community.general `deps`	Python packages	`with deps.declare("lib"):`
		`  import lib`
		
		`deps.validate(module)`
			`withStatement`
			`  .where(_.expression.isCallTo("deps.declare"))`
			`  .body.filter(_.isImport)`
			`  .map(_.importedName)`

`get_bin_path`	OS packages, OS, Python packages	`bin = module.get_bin_path(` `  "binary_name")`	`callTo("get_bin_path")` `  .arguments(0).resolveString()`
community.general `CmdRunner`	OS packages	`runner = CmdRunner(` `  module, "binary_name")`	`callTo("CmdRunner")` `  .arguments(1).resolveString()`

common pattern in which developers import a Python package dynamically using the built-in `importlib` library.

A common pattern for OS packages is to use the Ansible-provided `get_bin_path` function, which takes the name of a binary and resolves it to an absolute path. If the binary cannot be found, it either returns an empty value or raises an exception, depending on its arguments. These results can be inspected to check whether a binary exists on the system, and is thus often used to check for OS packages and platforms.

Finally, we find two patterns specific to the *community.general* collection, a prominent collection in our dataset. This collection offers two utilities to interact with dependencies. The `deps` utility offers an abstraction on top of the *guarded import* pattern using a context manager, whereas the `CmdRunner` class is an abstraction to interact with system binaries which internally calls `get_bin_path`. Since their usage may be widespread across the *community.general* collection, we include them in the catalogue.

We do not find any implementation patterns for runtime versions, API platforms, or miscellaneous packages as these are either not checked, or rely on ad hoc implementations. We also note that in rare cases, a plugin may use multiple mechanisms to check the same dependency.

2) Failure patterns: Of the 174 plugin dependencies that are checked, in 166 cases (95%) the plugin simply fails if the dependency is not satisfied. We only found 5 cases in which the plugin automatically installs the dependency, and 3 cases in which the plugin proceeds with degraded functionality.

We recognise 3 patterns in the failure behaviour of plugins. For 105 dependencies, the plugin aborts through Ansible's `fail_json` function, which fails the module's execution and returns structured output describing the failure reason. In another 26 dependencies, the plugin raises its own exception, whereas in the last 35 cases, the plugin propagates an exception from one of the implementation patterns (e.g., an exception raised by a `get_bin_path` call).

Orthogonally, to create an error message, the plugin can use Ansible's `missing_required_lib` function, which takes the name of a dependency and constructs an error message explaining the missing dependency and how it can be installed. However, we found it is used in only a third of all failure behaviours. Specifically, we observe such a call with 54 instances of the `fail_json` pattern, and with 3 instances of the exception raising pattern.

Summary: Most (79%) plugins check that their dependencies are satisfied. We discern 5 implementation patterns that are indicative of dependencies on Python or OS packages.

V. Automated Software Composition Analysis

RQ_1 showed that dependency types are diverse and that the documentation that specifies dependencies is informal and unstructured, hindering the ability to automatically extract dependencies from documented requirements. Having identified 5 dependency management patterns in RQ_2, we investigate whether the dependencies of a plugin can be extracted automatically from its implementation instead (RQ_3). To this end, we create a Software Composition Analysis (SCA) based on the Joern framework [12]. The analysis operates in two phases. First, it matches the patterns against individual functions in a collection's source code. However, these patterns may occur in common utility functions instead of the plugin implementation itself (cf. RQ_2). Therefore, in the second phase, the SCA propagates the identified dependencies to each function's transitive callers, thereby propagating the dependencies from common implementations to each plugin that uses them.

A. Semantic matching of dependency management patterns

We implement CPG queries for each of the 5 patterns, which the last column of Table II summarises as Scala-like pseudocode. As the concrete implementation of a pattern can vary in source code, the queries rely on semantic information such as data-flow information to overcome the limitations of purely syntactic pattern matching.

The query for *guarded imports* finds try-except blocks that contain an import, and extracts the variables assigned in the

block. It then uses data-flow information to find usages of those variables inside of conditions (e.g., in an if-statement), indicating conditional execution based on the result of the import. Finally, it marks the function containing the condition with the dependency names extracted from the import.

The query for the *community.general* `deps` pattern identifies `with` statements using the `deps.declare` function. It marks the functions containing a call to `deps.validate` with the dependency names extracted from the import statements in the with-block.

The queries for the *dynamic import*, `get_bin_path`, and *community.general* `CmdRunner` patterns all search for calls to the respective function. Functions containing such calls are marked with the dependency name obtained from the call's arguments. When the argument is a string literal, the name can be extracted straightforwardly. However, the argument can also be a variable reference, in which case the query attempts to resolve this reference to a single constant definition of a string literal. In case the reference cannot be resolved, resolves to multiple possible definitions, or is not a string literal, we cannot confidently extract the dependency name and therefore under-approximate by omitting the call.

B. Match propagation

Because multiple plugins may use the same utility functions (cf. RQ_2), dependencies in such functions need to be propagated to the plugins that use them. Therefore, we transitively propagate the dependencies backwards along the call graph to the function's callers. Due to Python being dynamically typed, the call graph constructed by Joern is an approximation. Specifically, if Joern cannot infer type information for the receiver of a method call, it resolves calls by name. This can lead to a vast over-approximation for generic method names like "run", which risks generating many false positives. Therefore, our propagation mechanism is conservative, and only propagates dependencies if the callee's receiver type is known, or if the function name is unique in the project.

We also noticed several shortcomings in Joern's call graph construction for object-oriented Python, such as missing call graph edges for super calls and self calls. As these limitations hamper our dependency propagation, we implement additional post-processing passes on the CPG to add the missing edges.

C. RQ_3: To what extent can Ansible plugin dependencies be identified automatically?

The ground truth, constructed from the manually-investigated sample of valid dependency-specifying documented requirements of RQ_2, comprises 221 dependencies spread across 166 plugins. Running the Software Composition Analysis for these plugins results in 300 dependencies for 127 plugins. 57 of the results reported by the SCA are individual parts of the same dependency, e.g., individual Python packages such as `win32pipe` and `win32event`, both belonging to the `pywin32` Python dependency. We aggregate these 57 results into 17 groups, as described in Section III-D, leading to 260 unique extracted dependencies.

TABLE III: Recall per dependency type

Dependency type	Lower bound		Upper bound	
Package: Python	79.3%	111/140	87.4%	111/127
Package: OS	45.2%	19/42	57.6%	19/33
Package: Misc	0.00%	0/2	0.00%	0/2
Platform: OS	30.8%	4/13	57.1%	4/7
Platform: API	0.00%	0/7	0.00%	0/5
Runtime: Python	6.25%	1/12	50.0%	1/2
Runtime: Ansible	0.00%	0/1	*n/a*	0/0

```
IPTABLES = {'ipv4': 'iptables', 'ipv6': 'ip6tables'}
module.get_bin_path(IPTABLES[ip_version], True)
```

Fig. 5: A missed dependency due to complex data flow.

1) Recall: The SCA correctly identified 135 dependencies out of the 221 unique dependencies in the entire ground truth, leading to a lower bound on recall of 61.09%. However, considering only those dependencies for which a check is implemented, the ground truth size decreases to 176, leading to an upper bound on recall of 76.7%. Table III depicts lower and upper bounds for individual dependency types. Python packages, the most common dependency type, can be identified with high recall, while around half of the OS packages can be identified. Other dependency types are often managed ad hoc and thus cannot be detected with high recall.

We investigated the root causes of false negatives in more detail. 13 false negatives were caused by the SCA failing to match a dependency pattern because the dependency was checked with an ad hoc implementation. For another 8 false negatives, the SCA failed to match their code against a pattern because the data flow was too complex. Figure 5 provides an example of an implementation that was not matched because the argument to `get_bin_path` could not be resolved to a single literal definition. In another 12 cases, the SCA matched the implementation with a pattern but failed to propagate the dependency through the call graph due to a lack of type information on the receiver objects of method calls. Further improvements to Joern's type inference could likely alleviate this issue without requiring changes to our approach.

2) Precision: Across the 260 grouped results, we find 193 non-built-in true positives, and 27 built-in true positives. This corresponds to a precision between 74.2% and 84.6%. We also find 30 dependencies (13.7%) that are managed by the plugin but not mentioned in the documentation.

We again identify root causes for false positives. The most common root cause (23 false positives) occurs only in the *ansible.builtin* collection. As its implementation is part of Ansible itself, the SCA inspects the entire Ansible implementation. This caused it to extract Ansible's own dependencies, which it then erroneously propagated to the built-in plugins, although they are not dependencies of the plugins themselves. In practical applications, omitting *ansible.builtin* would raise precision to between 89.5% and 95.0%.

Summary: Dependency management patterns can be used to automatically extract plugin dependencies, with recall between 61% and 77%, and precision between 74% and 95%. The dependency SCA complements the plugin documentation by identifying 30 undocumented dependencies.

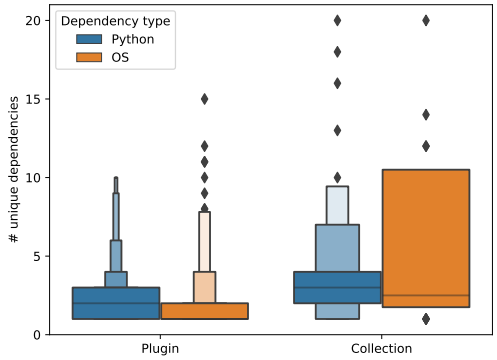

Fig. 6: Distributions of the number of unique extracted dependencies for plugins and collections with at least one dependency pattern for a given type. An extreme outlier caused by the *community.general* collection has been omitted.

D. RQ_4: How prevalent are Ansible plugin dependencies in the wild?

To investigate the dependencies of Ansible plugins at scale, we applied the Software Composition Analysis on all 11,241 plugins from 187 of the 188 collections in the dataset. We exclude plugins from *ansible.builtin* as they caused too many false positives in RQ_3 due to the SCA reporting Ansible's own dependencies. To map dependency management patterns to dependency types, we consider usages of *guarded imports*, *dynamic imports*, and *community.general* `deps` to signify Python packages, whereas `get_bin_path` and *community.general* `CmdRunner` signify OS packages.

Across the 11,241 plugins, we identify dependencies in 38% of plugins, meaning many plugins depend on other software. Specifically, 35.9% depend on Python libraries, 1.8% depend on OS packages, and 0.2% (30 plugins) depend on both. When aggregating all plugins in a collection, we find that 51% of the collections have at least one dependency. 40.6% of the collections depend on Python libraries, 6 collections (3.2%) depend on OS packages, and 14 collections (7.5%) depend on both. The large difference in the number of identified Python library and OS package dependencies is consistent with our qualitative findings in RQ_1, especially considering that many OS packages are either not checked, or checked ad hoc.

Figure 6 depicts the distribution of the number of unique Python library and OS package dependencies for plugins and collections. Each plot considers only plugins or collections with at least one dependency of the given type. We observe that the majority of plugins and collections depend on multiple Python libraries, sometimes as many as 10 individual libraries in a single plugin. Plugins generally depend on fewer OS packages, yet of the 20 collections that depend on OS packages, 19 depend on multiple. However, the SCA may report multiple binaries of the same OS package separately (cf. RQ_3) which we cannot group automatically, so the number of OS packages required may be lower in practice. Finally, the *community.general* collection forms an extreme outlier (hidden in Figure 6), as it depends on 126 Python packages and 205 OS binaries. It was created to house many unrelated plugins that were migrated out of Ansible's core codebase, of which many have their own unique dependencies.

These results show that dependencies originating from other ecosystems are common among plugins and collections. This establishes extensive deployment software supply chains for Ansible code that relies on such plugins and collections.

Summary: 38% of plugins and 51% of collections have dependencies. Plugins often depend on several Python libraries, yet rarely on multiple OS packages. The SCA enables large-scale analysis of deployment software supply chains.

VI. Discussion

In this section, we discuss the implications of our findings.

A. Towards automated deployment SBOM generation

Generating SBOMs for deployment code could aid practitioners in managing their dependencies and those of the IaC plugins they use, for instance to patch security vulnerabilities, update outdated dependencies, and verify licence compliance [6]. Such SBOMs could be distributed alongside every Ansible collection to document its components. Similarly, Ansible practitioners could leverage deployment SBOMs to gain a better understanding of their deployment software supply chains. The taxonomy from RQ_1 serves as a foundation to decide on the types and properties of software dependencies represented in deployment SBOMs. It shows that they need to **support various types of third-party dependencies**, including traditional dependencies (e.g., Python packages), but also OS platforms and remote APIs. Moreover, deployment SBOMs would need to **distinguish between the machines on which the components are installed**, such as IaC controllers or configured hosts.

The SCA presented in Section V serves as a stepping stone towards automated generation of such deployment SBOMs. The results of RQ_3 show that it can achieve high recall for the most common dependency types, and that it can identify undocumented dependencies. However, note that the SCA cannot directly generate SBOMs, as SBOMs document what is *installed*, whereas the SCA instead reports what is *required*. Nonetheless, it could be applied to **generate preliminary deployment SBOMs** to be augmented manually by practitioners. Moreover, it could be used to **enrich existing SBOMs**, for instance, by tagging components that are installed for the deployment code. Such information could be used to verify whether deployment dependencies are satisfied, or to debloat unused dependencies [13].

Finally, our taxonomy and SCA only consider dependencies in Ansible's runtime layer. To target complete deployment SBOMs, they could be extended with application-layer dependencies, such as Ansible roles [14] and the plugins themselves. As these dependencies are specified in well-defined, machine-readable requirements files [8], such extensions could be made straightforwardly.

B. Implications for Ansible practitioners

The results of RQ_1 show that documented dependencies are informal and unstructured, whereas in RQ_2, we found that plugins rarely automatically install dependencies or check their versions. This may hamper the reliability and reproducibility of Ansible projects. For instance, recent work identified missing dependencies as the main cause of crashes in automatically-generated Ansible code [15]. Moreover, changes to plugin dependencies have already caused real-world incompatibilities in Ansible code that uses the plugins [16], [17].

Therefore, we recommend practitioners to **adopt dependency management patterns** such as those presented in RQ_2 and to thoroughly check the versions of the installed dependencies. Moreover, we recommend the community to adopt **automated dependency management** to avoid dependency-related issues. For instance, Ansible provides *execution environments* [18] that use containerisation to create reproducible environments for IaC controllers. However, for remote hosts, other IaC dependency management solutions are required. For instance, future work could investigate adding support within Ansible to specify dependencies and install them automatically. Finally, we urge Ansible plugin maintainers to adopt **structured, consistent, and machine-readable dependency specifications** that lend themselves well to automated analysis. To this end, our taxonomy may aid in designing such specifications, whereas our SCA may be used to semi-automatically generate them from the plugins' implementations.

C. Other applications of the Software Composition Analysis

Aside from generating deployment SBOMs, the SCA created and evaluated in RQ_3 could form the basis for practical tooling for Ansible developers. For instance, it could be used to automatically report which dependencies are necessary to run an Ansible script. For plugin developers, the technique could be leveraged to identify undocumented dependencies or missing dependency management patterns.

The SCA could also be applied to study the evolution of Ansible collection dependencies. We conducted a preliminary study suggesting that nearly 40% of plugins and 86% of collections with dependency management patterns have undergone changes to the patterns. However, we observed little to no co-evolution with the documented requirements. A manual investigation of a sample of these changes also revealed that changes to the patterns are often refactorings, such as developers introducing a dependency management pattern for a previously-unchecked dependency. This preliminary experiment suggests that dependencies may evolve over time, which may bring about challenges for end users, but that the documentation may also be outdated or incorrect. Future work could study dependency changes in more depth.

D. Limitations of the Software Composition Analysis

In RQ_3, we found that our extractor can identify Python library dependencies accurately. However, it struggles to extract other dependency types and unchecked dependencies.

For unchecked Python libraries, one strategy may be to consider *all* import statements, including unguarded ones. However, this may extract many trivial built-in packages provided either by Ansible or Python itself, causing high false positive rates. Moreover, one would need to discern between imports of first-party and third-party code.

For OS packages, the extractor identifies implementation patterns used to check for the presence of a binary, causing it to miss unchecked OS dependencies whose binaries are used directly. Therefore, it may be possible to identify unchecked dependencies by also matching such direct usages.

We found no implementation patterns for Python runtime versions (cf. RQ_2) and thus fail to identify many of those dependencies. Future work could investigate whether other techniques could identify required Python versions, e.g., inspecting syntax features or imported built-in packages [19].

The `missing_required_lib` function (cf. RQ_2) may be useful to identify dependencies that are managed through ad hoc implementations. However, we note that it is used in only a third of the dependency management implementations, and that its presence does not convey the dependency types.

Finally, our SCA cannot identify version constraints from the plugin implementations. Future work could consider inspecting the usages of identified Python packages to identify compatible version ranges [20].

VII. Threats to Validity

As a threat to *internal* validity, the manual investigation performed in RQ_1 and RQ_2 was performed by a single labeller, which may lead to subjective bias. We mitigated this threat by limiting ourselves to purely objective observations and consulting online documentation in case of ambiguity. Moreover, we avoid subjectivity in RQ_3 by providing a lower and upper bound on precision and recall. Furthermore, the manually investigated sample may not generalise to the entire studied dataset. We mitigate this threat by applying established filtering criteria when constructing the dataset, choosing a statistically significant sample size, and sampling from a deduplicated pool to avoid studying duplicates.

A threat to *construct* validity stems from the reliance of the Software Composition Analysis on instances of dependency management implementation patterns as a proxy for dependencies. The SCA and Joern, the framework on which it is built, may suffer from technical bugs and limitations (cf. Section VI-D) which may hamper the correct identification of dependencies. We mitigated this threat by evaluating the SCA's precision and recall in RQ_3, which showed it is accurate. Moreover, we omitted the *ansible.builtin* collection to avoid studying Ansible's own dependencies.

As a threat to *external validity*, our study focuses only on Ansible. Nonetheless, we believe that our taxonomy may apply to other configuration management IaC tools, such as Chef and Puppet. Future work could replicate our study to identify the types and properties of their plugins' dependencies. Moreover, both Puppet and Chef offer fine-grained specifications of plugin dependencies, which we believe would facilitate

automated analysis of their dependency networks. Future work could also replicate our study to other types of IaC tools, such as provisioning tools (e.g., Terraform, Pulumi), which likely make intense use of remote APIs.

VIII. Related Work

1) Run-time and development dependencies: Many studies have investigated dependencies that are necessary at run time or during development. These studies typically focus on the ecosystems formed around dependency management tools. For instance, the NPM ecosystem for JavaScript has been widely studied, often in a security context [21]–[27]. Other work has focused on ecosystems like Java's Maven [28]–[30], Python's PyPI [31]–[33], C/C++ [34], OS packages [35], or comparing common properties across multiple ecosystems [36]–[39]. Other researchers have instead studied dependencies in specific software domains. Huang et al. [40] study dependency-related bugs in Deep Learning stacks, whereas Fang et al. [41] identify dependency antipatterns in distributed microservice architectures. Our work differs from these existing studies as we investigate deployment dependencies in Ansible plugins, for which no structured manifests exist.

2) Deployment dependencies: In contrast to the work on run-time and development dependencies outlined above, recent work has investigated deployment dependencies, which typically manifest themselves in IaC specifications.

For instance, the ecosystem of Docker images has been the subject of several empirical studies. Zhao et al. [42] investigate Docker Hub's image characteristics, such as layer sizes. Zerouali et al. investigate the use of run-time dependencies from ecosystems like NPM, PyPI, and Debian packages in Docker images [43]–[46], focusing on security vulnerabilities and outdatedness. Lin et al. [47] study the evolution of Docker files and their corresponding images on Docker Hub. Ibrahim et al. [48] compare characteristics such as image size between official and community images on Docker Hub. Opdebeeck et al. [49] investigate deployment dependencies in Docker images that result from inheritance between images. Zerouali et al. [50] investigate the ecosystem forming around Helm charts for Kubernetes, focusing on outdatedness and security vulnerabilities in the Docker images they deploy.

Most closely related to our study is the one of Opdebeeck et al. [14], who study one part of deployment dependencies in the Ansible ecosystem, namely open-source "roles", which form reusable blueprints of infrastructure code. However, their study focuses on how these Ansible roles are versioned and evolve over time, rather than their dependencies. Moreover, they only consider application-layer dependencies, whereas our study investigates dependencies in Ansible's runtime layer.

3) Dependency identification: Many of the aforementioned studies rely on dependency manager manifest files to identify third-party software dependencies. However, some languages, notably C/C++, have no widely-used dependency management tools, posing a challenge to identify dependencies. Moreover, dependency manifests may not be available for software distributed in binary formats, e.g., mobile applications. Therefore, several earlier studies have investigated how dependencies can be identified for systems written in such languages.

Many of these prior approaches rely on code clone detection, either in source code [51]–[54] or in compiled binaries [55]–[58]. Tang et al. [59] combine such code clone detection techniques with information extracted from numerous package managers and build systems for C and C++. Our SCA differs from these existing approaches as we do not rely on code clone detection and instead match dependency management patterns in the source code.

4) SBOMs and Software Supply Chains: SBOMs can be created in various formats, of which SPDX [60] and CycloneDX [61] are the most commonly adopted [62]. They can be generated by numerous practical tools, such as Syft [63], Trivy [64], and Snyk [65]. Our work is complementary to these existing formats and approaches, as our taxonomy can guide extensions to SBOM formats, and our SCA enables augmentation of SBOMs generated by existing tools.

Recent work has investigated practical challenges related to SBOM adoption, both through developer surveys [62], [66] and discussions on development platforms [67]. Moreover, Nocera et al. [68] found that SBOM adoption in GitHub projects is low but increasing. Finally, prior work has compared and evaluated different SBOM generators, finding that they may produce inconsistent results [69], [70]. Our work differs as we study the elements needed to create SBOMs for Infrastructure as Code, and how this information can be extracted.

IX. Conclusion

Infrastructure as Code tools form integral parts of modern software supply chains. IaC projects may themselves be supported by *deployment software supply chains*, but these cannot yet be analysed automatically. Therefore, we performed a qualitative study into how an automated analysis could be designed for Ansible IaC plugins. Based on a manual analysis of 266 documented plugin requirements, we constructed a taxonomy of 7 dependency types that deployment SBOMs would need to support, including development libraries and OS packages, yet also remote APIs and platforms. Moreover, we identified that deployment SBOMs would need to distinguish between the machines on which these dependencies are installed. As Ansible plugin requirements are specified informally and without structure, we designed a Software Composition Analysis based on 5 manually-identified dependency implementation patterns. Our SCA can identify the most common dependency types from plugin implementations, achieving 61%–77% recall and 74%–95% precision. Applying our SCA at scale, we found that 38% of plugins and 51% of plugin collections have dependencies. Our taxonomy can guide the design of deployment SBOMs, whereas the SCA can be used to augment existing SBOMs with deployment-related information.

Acknowledgements

This research was partially funded by the "Cybersecurity Initiative Flanders" project and the Research Foundation Flanders (FWO) under Grant No. 1SD4321N and V431423N.

REFERENCES

[1] K. Morris, *Infrastructure as Code: Managing Servers in the Cloud*, 1st ed. O'Reilly, 2016.

[2] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, "Adoption, support, and challenges of infrastructure-as-code: Insights from industry," in *Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution, Industrial Track*, ser. ICSME '19, 2019, pp. 580–589.

[3] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment – 17th International Conference, Proceedings*, 2020, pp. 23–43.

[4] Executive order 14028: Improving the nation's cybersecurity. [Online]. Available: https://www.federalregister.gov/executive-order/14028

[5] The council agrees to strengthen the security of ict supply chains. [Online]. Available: https://www.consilium.europa.eu/en/press/press-releases/2022/10/17/the-council-agrees-to-strengthen-the-security-of-ict-supply-chains/

[6] S. Hendrick. (2022) The state of software bill of materials (sbom) and cybersecurity readiness. [Online]. Available: https://www.linuxfoundation.org/research/the-state-of-software-bill-of-materials-sbom-and-cybersecurity-readiness

[7] StackExchange, Inc. (2023) 2023 annual stackoverflow developer survey. [Online]. Available: https://survey.stackoverflow.co/2023/#section-most-popular-technologies-other-tools

[8] Ansible. Installing multiple roles from a file. [Online]. Available: https://docs.ansible.com/ansible/latest/galaxy/user_guide.html#installing-multiple-roles-from-a-file

[9] G.-P. Drosos, T. Sotiropoulos, G. Alexopoulos, D. Mitropoulos, and Z. Su, "When your infrastructure is a buggy program: Understanding faults in infrastructure as code ecosystems," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, p. 359, 2024.

[10] Ansible. Handling import errors. [Online]. Available: https://docs.ansible.com/ansible/latest/dev_guide/testing/sanity/import.html#handling-import-errors

[11] W. G. Cochran, *Sampling Techniques*, 3rd ed. John Wiley & Sons, 1977.

[12] joern.io, "Joern: The Bug Hunter's Workbench," Jan. 2024. [Online]. Available: https://github.com/joernio/joern

[13] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, "A comprehensive study of bloated dependencies in the maven ecosystem," *Empir. Softw. Eng.*, vol. 26, no. 3, p. 45, 2021. [Online]. Available: https://doi.org/10.1007/s10664-020-09914-8

[14] R. Opdebeeck, A. Zerouali, C. Velázquez-Rodríguez, and C. D. Roover, "On the practice of semantic versioning for Ansible Galaxy roles: An empirical study and a change classification model," *J. Syst. Softw.*, vol. 182, p. 111059, 2021.

[15] M. M. Hassan, J. Salvador, S. K. K. Santu, and A. Rahman, "State reconciliation defects in infrastructure as code," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1865–1888, Jul. 2024.

[16] J. Tölle. Hetzner.hcloud issue: Check installed version of hcloud-python. [Online]. Available: https://github.com/ansible-collections/hetzner.hcloud/issues/211

[17] J. Geerling. Ansible issue: docker_image module typeerror. [Online]. Available: https://github.com/ansible/ansible/issues/35612

[18] Ansible. Getting started with execution environments. [Online]. Available: https://docs.ansible.com/ansible/latest/getting_started_ee/index.html

[19] M. Gerhold, L. Solovyeva, and V. Zaytsev, "The limits of the identifiable: Challenges in python version identification with deep learning," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2024, pp. 137–146.

[20] A. Zerouali, C. Velázquez-Rodríguez, and C. De Roover, "Identifying versions of libraries used in stack overflow code snippets," in *Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories*. IEEE, 2021, pp. 341–345.

[21] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, and X. Peng, "Demystifying the vulnerability propagation and its evolution via dependency trees in the NPM ecosystem," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 672–684.

[22] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the JavaScript package ecosystem," in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, M. Kim, R. Robbes, and C. Bird, Eds. ACM, 2016, pp. 351–361.

[23] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, A. Zaidman, Y. Kamei, and E. Hill, Eds. ACM, 2018, pp. 181–191.

[24] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. M. González-Barahona, "An empirical analysis of technical lag in npm package dependencies," in *New Opportunities for Software Reuse - 17th International Conference, ICSR 2018, Madrid, Spain, May 21-23, 2018, Proceedings*, ser. Lecture Notes in Computer Science, R. Capilla, B. Gallina, and C. Cetina, Eds., vol. 10826. Springer, 2018, pp. 95–110.

[25] M. Alfadel, D. E. Costa, E. Shihab, and B. Adams, "On the discoverability of npm vulnerabilities in node.js projects," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, pp. 91:1–91:27, 2023.

[26] M. A. R. Chowdhury, R. Abdalkareem, E. Shihab, and B. Adams, "On the untriviality of trivial packages: An empirical study of npm JavaScript packages," *IEEE Trans. Software Eng.*, vol. 48, no. 8, pp. 2695–2708, 2022.

[27] B. Rombaut, F. R. Côgo, B. Adams, and A. E. Hassan, "There's no such thing as a free lunch: Lessons learned from exploring the overhead introduced by the Greenkeeper dependency bot in npm," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, pp. 11:1–11:40, 2023.

[28] K. Huang, B. Chen, C. Xu, Y. Wang, B. Shi, X. Peng, Y. Wu, and Y. Liu, "Characterizing usages, updates and risks of third-party libraries in Java projects," *Empir. Softw. Eng.*, vol. 27, no. 4, p. 90, 2022.

[29] H. He, R. He, H. Gu, and M. Zhou, "A large-scale empirical study on Java library migrations: prevalence, trends, and rationales," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 478–490.

[30] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning versus breaking changes: A study of the maven repository," in *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014*. IEEE Computer Society, 2014, pp. 215–224.

[31] W. Guo, Z. Xu, C. Liu, C. Huang, Y. Fang, and Y. Liu, "An empirical study of malicious code in PyPI ecosystem," in *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 2023, pp. 166–177.

[32] W. Xu, H. He, K. Gao, and M. Zhou, "Understanding and remediating open-source license incompatibilities in the PyPI ecosystem," in *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 2023, pp. 178–190.

[33] Y. Wang, M. Wen, Y. Liu, Y. Wang, Z. Li, C. Wang, H. Yu, S. Cheung, C. Xu, and Z. Zhu, "Watchman: monitoring dependency conflicts for Python library ecosystem," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 125–135.

[34] D. Wu, L. Chen, Y. Zhou, and B. Xu, "How do developers use C++ libraries? an empirical study," in *The 27th International Conference on Software Engineering and Knowledge Engineering, SEKE 2015, Wyndham Pittsburgh University Center, Pittsburgh, PA, USA, July 6-8, 2015*, H. Xu, Ed. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2015, pp. 260–265.

[35] R. Bajaj, E. Fernandes, B. Adams, and A. E. Hassan, "Unreproducible builds: time to fix, causes, and correlation with external ecosystem factors," *Empir. Softw. Eng.*, vol. 29, no. 1, p. 11, 2024.

[36] R. Abdalkareem, V. Oda, S. Mujahid, and E. Shihab, "On the impact of using trivial packages: an empirical case study on npm and PyPI," *Empir. Softw. Eng.*, vol. 25, no. 2, pp. 1168–1204, 2020.

[37] A. Zerouali, T. Mens, A. Decan, and C. D. Roover, "On the impact of security vulnerabilities in the npm and RubyGems dependency networks," *Empir. Softw. Eng.*, vol. 27, no. 5, p. 107, 2022.

[38] A. Decan, T. Mens, and M. Claes, "An empirical comparison of dependency issues in OSS packaging ecosystems," in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER*

2017, Klagenfurt, Austria, February 20-24, 2017, M. Pinzger, G. Bavota, and A. Marcus, Eds. IEEE Computer Society, 2017, pp. 2–12.

[39] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, J. M. González-Barahona, A. Hindle, and L. Tan, Eds. IEEE Computer Society, 2017, pp. 102–112.

[40] K. Huang, B. Chen, S. Wu, J. Cao, L. Ma, and X. Peng, "Demystifying dependency bugs in deep learning stack," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds. ACM, 2023, pp. 450–462.

[41] H. Fang, Y. Cai, R. Kazman, and J. Lefever, "Identifying anti-patterns in distributed systems with heterogeneous dependencies," in *20th International Conference on Software Architecture, ICSA 2023 - Companion, L'Aquila, Italy, March 13-17, 2023*. IEEE, 2023, pp. 116–120.

[42] N. Zhao, V. Tarasov, H. Albahar, A. Anwar, L. Rupprecht, D. Skourtis, A. S. Warke, M. Mohamed, and A. R. Butt, "Large-scale analysis of the Docker Hub dataset," in *2019 IEEE International Conference on Cluster Computing, CLUSTER 2019, Albuquerque, NM, USA, September 23-26, 2019*. IEEE, 2019, pp. 1–10.

[43] A. Zerouali, V. Cosentino, T. Mens, G. Robles, and J. M. González-Barahona, "On the impact of outdated and vulnerable Javascript packages in Docker images," in *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, X. Wang, D. Lo, and E. Shihab, Eds. IEEE, 2019, pp. 619–623.

[44] A. Zerouali, T. Mens, and C. D. Roover, "On the usage of *JavaScript*, *Python* and *Ruby* packages in Docker Hub images," *Sci. Comput. Program.*, vol. 207, p. 102653, 2021.

[45] A. Zerouali, T. Mens, G. Robles, and J. M. González-Barahona, "On the relation between outdated Docker containers, severity vulnerabilities, and bugs," in *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, X. Wang, D. Lo, and E. Shihab, Eds. IEEE, 2019, pp. 491–501.

[46] A. Zerouali, T. Mens, A. Decan, J. M. González-Barahona, and G. Robles, "A multi-dimensional analysis of technical lag in Debian-based Docker images," *Empir. Softw. Eng.*, vol. 26, no. 2, p. 19, 2021.

[47] C. Lin, S. Nadi, and H. Khazaei, "A large-scale data set and an empirical study of Docker images hosted on Docker Hub," in *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 2020, pp. 371–381.

[48] M. H. Ibrahim, M. Sayagh, and A. E. Hassan, "Too many images on DockerHub! how different are images for the same system?" *Empir. Softw. Eng.*, vol. 25, no. 5, pp. 4250–4281, 2020.

[49] R. Opdebeeck, J. Lesy, A. Zerouali, and C. D. Roover, "The Docker Hub image inheritance network: Construction and empirical insights," in *23rd IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2023, Bogotá, Colombia, October 2-3, 2023*, L. Moonen, C. D. Newman, and A. Gorla, Eds. IEEE, 2023, pp. 198–208.

[50] A. Zerouali, R. Opdebeeck, and C. D. Roover, "Helm charts for Kubernetes applications: Evolution, outdatedness and security risks," in *20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15-16, 2023*. IEEE, 2023, pp. 523–533.

[51] S. Woo, S. Park, S. Kim, H. Lee, and H. Oh, "Centris: A precise and scalable approach for identifying modified open-source software reuse," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 860–872.

[52] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "Déjàvu: a map of code duplicates on GitHub," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 84:1–84:28, 2017.

[53] J. Wu, Z. Xu, W. Tang, L. Zhang, Y. Wu, C. Liu, K. Sun, L. Zhao, and Y. Liu, "OSSFP: precise and scalable C/C++ third-party library detection using fingerprinting functions," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 270–282.

[54] L. Jiang, H. Yuan, Q. Tang, S. Nie, S. Wu, and Y. Zhang, "Third-party library dependency for large-scale SCA in the C/C++ ecosystem: How far are we?" in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, R. Just and G. Fraser, Eds. ACM, 2023, pp. 1383–1395.

[55] C. Yang, Z. Xu, H. Chen, Y. Liu, X. Gong, and B. Liu, "Modx: Binary level partial imported third-party library detection through program modularization and semantic matching," *CoRR*, vol. abs/2204.08237, 2022.

[56] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying open-source license violation and 1-day security risk at large scale," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 2169–2185.

[57] W. Tang, D. Chen, and P. Luo, "Bcfinder: A lightweight and platform-independent tool to find third-party components in binaries," in *25th Asia-Pacific Software Engineering Conference, APSEC 2018, Nara, Japan, December 4-7, 2018*. IEEE, 2018, pp. 288–297.

[58] C. Soh, H. B. K. Tan, Y. L. Arnatovich, A. Narayanan, and L. Wang, "Libsift: Automated detection of third-party libraries in android applications," in *23rd Asia-Pacific Software Engineering Conference, APSEC 2016, Hamilton, New Zealand, December 6-9, 2016*, A. Potanin, G. C. Murphy, S. Reeves, and J. Dietrich, Eds. IEEE Computer Society, 2016, pp. 41–48.

[59] W. Tang, Z. Xu, C. Liu, J. Wu, S. Yang, Y. Li, P. Luo, and Y. Liu, "Towards understanding third-party library dependency in C/C++ ecosystem," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 106:1–106:12.

[60] Spdx. [Online]. Available: https://spdx.dev/

[61] Cyclonedx. [Online]. Available: https://cyclonedx.org/

[62] B. Xia, T. Bi, Z. Xing, Q. Lu, and L. Zhu, "An empirical study on software bill of materials: Where we stand and the road ahead," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 2630–2642.

[63] Anchore. Syft. [Online]. Available: https://github.com/anchore/syft/

[64] Trivy. [Online]. Available: https://aquasecurity.github.io/trivy/v0.56/docs/supply-chain/sbom/

[65] Snyk. [Online]. Available: https://snyk.io/solutions/software-supply-chain-security/

[66] T. Stalnaker, N. Wintersgill, O. Chaparro, M. Di Penta, D. M. German, and D. Poshyvanyk, "Boms away! inside the minds of stakeholders: A comprehensive study of bills of materials for software systems," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623347

[67] T. Bi, B. Xia, Z. Xing, Q. Lu, and L. Zhu, "On the way to sboms: Investigating design issues and solutions in practice," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 6, Jun. 2024. [Online]. Available: https://doi.org/10.1145/3654442

[68] S. Nocera, S. Romano, M. D. Penta, R. Francese, and G. Scanniello, "Software bill of materials adoption: A mining study from github," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2023, pp. 39–49.

[69] M. F. Rabbi, A. I. Champa, C. Nachuma, and M. F. Zibran, "Sbom generation tools under microscope: A focus on the npm ecosystem," in *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1233–1241. [Online]. Available: https://doi.org/10.1145/3605098.3635927

[70] S. Yu, W. Song, X. Hu, and H. Yin, "On the correctness of metadata-based sbom generation: A differential analysis approach," in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2024, pp. 29–36.