A Study into the Use of Version Locking in Gradle Projects

Joppe Boerop[1]

Supervisor(s): Sebastian Proksch[1]**, Cathrine Paulsen**[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Joppe Boerop
Final project course: CSE3000 Research Project
Thesis committee: Sebastian Proksch, Cathrine Paulsen, Georgios Iosifidis

Abstract

Developers often use dependency managers to make updating dependencies easier. These dependency managers allow permissive declaration strategies to be used which automatically keep dependencies up-to-date. To prevent these automatic updates from breaking projects, developers can use version locking to lock specific versions in place. To investigate how version locking is used we have done research into version locking in projects using Gradle. We found that version locking is not widely adapted, as only 0.34% of our sampled Gradle projects contained a lock file. Our analysis into the use of version ranges in projects using version locking showed that only 29.5% to 44.4% of analyzed projects using version locking, used version ranges. This is surprising as version locking is most effective when using version ranges. Our research into the effect of version locking showed that in 18.2% of the analyzed projects, version locking prevented the build from failing. Looking into the negative effects of version locking, we found that 9.8% of the dependencies are at risk of being a vulnerability, as for these dependencies there are newer versions available which might introduce security patches.

1 Introduction

In software development, developers often use open-source code in their projects to avoid having to reinvent the wheel and instead use well-documented and well-maintained code for functionality that is not specific to their own project. Managing these dependencies can become difficult, as dependencies might use other dependencies themselves. These transitive dependencies may intersect with other dependencies used in the project, leading to version conflicts. Dependency managers, such as Gradle, are tools designed to make managing these dependencies easier by automatically resolving and installing dependencies. In most cases, dependencies continue to be improved. Security issues may be discovered, for example, which means a new version needs to be published to fix the issue. It is therefore important to frequently update these dependencies. Gradle allows developers to use permissive version declarations to automatically use the latest version of a dependency. However, sometimes these automatic updates can break a project. This happens when an update contains so-called breaking changes: changes to the interface of the dependency which can break the project using it. Therefore, Gradle recommends the use of "version locking" when using permissive declaration strategies [1]. When version locking is used, a lock file is generated, listing the specific version used for every dependency. After this, during the build of a project, Gradle uses the version listed in the lock file instead of resolving the dependencies again, leading to reproducible builds and preventing the project from unexpectedly breaking.

Semantic versioning was introduced as a way for developers to easily see if an update introduces breaking changes [2]. In a version like X.Y.Z, X is called the major version, Y the minor version and Z the patch version. According to semantic versioning, only major version updates should introduce breaking changes. However, research has shown that this convention is often not followed and breaking changes are also introduced in other updates [3; 4; 5]. This means that developers cannot rely on semantic versioning to guarantee reproducible builds, demonstrating the need for version locking when using permissive declaration strategies. Research by Pashchenko et al. [6] confirms that the fact that updates may introduce breaking changes is one of the key reasons developers decide not to update their dependencies. Version locking could be useful here, as it would allow developers to use permissive declaration strategies, while still guaranteeing reproducible builds. A recent study by Gamage et al. [7] showed, however, that version locking is rarely used in Gradle. They found that only 0.3% to 0.9% of the 323 projects analyzed used version locking. In this paper, we will extend this research by verifying the found adoption rate for a much larger sample set and looking into the effects of the use of version locking.

The research questions that will be investigated in this paper are:

RQ1. How widespread is the use of version locking?
Version locking is a built-in feature in Gradle, but how often is it actually used? Is it widely adopted or are there only a few developers who use it? We sampled over 12,000 Gradle projects from GitHub and found that only 0.34% of them used version locking.

RQ2. How widespread is the use of version ranges in projects that also use version locking?
Version locking is most useful when using version ranges. If version ranges are not used, direct dependencies are pinned to a specific version anyway and only transitive dependencies can benefit from version locking. To find out if version ranges are indeed used often in projects with version locking, we sampled 47 projects using version locking and analyzed how many of them used version ranges. Surprisingly, we found that this was the case for only 29.5% to 44.4% of these projects.

RQ3. How can we measure the effect of version locking?
The effect of version locking can be positive and negative. Positive effects include not breaking a project when breaking changes are introduced in newer dependency versions. Negative effects could occur when a newer version of a dependency is released, but not used because the version is locked to an older version. This version could, for example, introduce security patches, which are now not adopted in the project. To analyze the positive effects, we tried to build the projects that we sampled after removing the lock files. For 18.2% of the projects, version locking prevented the build from failing, showing the need for version locking in these projects. For the negative effects, we analyzed how often projects had locked dependencies to an older version and how big the difference between this older version and the latest version was. We saw that 41.4% of the projects used older versions of dependencies. Furthermore, we saw that at most 9.8% of the dependencies used throughout the analyzed projects contain vulnerabilities.

For our research we wrote several Python scripts to analyze data. These scripts can be found in our GitHub repository [8].

It is worth noting that our original intention was to investigate version locking in Maven. Version locking is not a built-in feature in Maven, meaning a plugin is needed. We identified a plugin and investigated its adoption, but found that it was used so rarely that a full research into projects using it would be unfeasible. As we were unable to find another, more frequently used plugin, we decided to switch to another dependency manager. We chose Gradle as it uses Maven Central for its dependencies, just like Maven. As version locking is a built-in feature in Gradle, we expected that we would be able to find a sufficient number of projects using version locking when looking at Gradle projects.

2 Dependency Management in Gradle

Gradle provides a dependency management tool, which helps developers by automating the resolution and installation of dependencies. This section will give a short explanation of the features that are relevant to this paper.

2.1 Dependency Declaration

Dependencies are declared in a build file (`build.gradle`) written in either Kotlin or Groovy. A dependency declaration consists of the dependency's group, name and version and can be written in the following ways:

- String notation: `"group:name:version"`
- Map notation: `group = "group", name = "name", version = "version"`
  (Note that this is the map notation written in Kotlin. In Groovy, the =-signs are replaced by colons)
- Centralized dependencies: `"libs.name"`
  Centralizing dependencies makes declaring dependencies easier for projects with multiple build files. Specifying the versions of dependencies is done in a separate `libs.versions.toml` file using the map notation. In the `build.gradle` file you reference this version by writing `"libs.name"` where `name` is the name you gave to the version declaration in the `libs.versions.toml` file.

The version can either be a specific version, in which case the dependency will be pinned to that version, or a version range.

2.2 Version Ranges

In Gradle, version ranges can be declared using 3 methods:

- Using square brackets and parentheses. Square brackets are inclusive bounds and parentheses are exclusive bounds. Examples:
  - `[1.3, 1.8]`
    Includes all versions from version 1.3 up to and including version 1.8
  - `[1.3, 1.8)`
    Includes all versions from version 1.3 up to (but not including) 1.8
  - `[1.3, 1.8[`
    The same as above. [can be used as an alternative for). Similarly,] can be used as an alternative for (.
- Using prefix notation. By putting a plus sign (+) after the first part of a version, all versions with that first part will be included. Examples:
  - `1.3.+`
    Includes 1.3.0, 1.3.1, 1.3.2, etc.
  - `2.+`
    Includes 2.1, 2.2.4, 2.5, etc.
  - `+`
    Includes all versions. Similar to using latest notation (see below)
- Using latest notation. When using latest notation, Gradle will resolve the newest version of this dependency. There are two different latest notations:
  - `latest.version`
    Includes all released versions
  - `latest.integration`
    Includes all versions, including SNAPSHOT versions. SNAPSHOT versions are versions that are still under development, which are not guaranteed to be stable and could still be subject to change [9].

2.3 Version locking

Version locking in Gradle can be activated in the `build.gradle` file [1]. Locking all configurations is done by adding a `dependencyLocking` block:

```
dependencyLocking {
    lockAllConfigurations()
}
```

Specific configurations can be locked with:

```
configurations {
    compileClasspath {
        resolutionStrategy
            .activateDependencyLocking()
    }
}
```

After locking has been activated, a lock file needs to be generated. This is done by running the command `gradle dependencies --write-locks`. Only when a lock file is present and locking is activated, the locked versions will be enforced.

3 Data Collection

This section explains how we created our datasets for the different research questions

3.1 General Dataset of Gradle Projects

For RQ1 we looked into how often version locking is used in Gradle projects. For this, we needed a dataset of Gradle projects so we could investigate how many of those use version locking. To this end we sampled Java GitHub projects using a repository mining tool [10]. We decided on the following filters for the mining tool:

- Minimum number of stars: 10
- Last commit after: 01-01-2024

The minimum number of stars were chosen to filter out toy-projects. We believe these projects would skew our results as most of them are created for experimentation purposes and might therefore not reflect real-world dependency management practices. We excluded projects for which the last commit was before 01-01-2024 because we were interested in how much version locking is used at the current moment.

This resulted in a dataset of 29,210 repositories, which can be found in our GitHub repository at `data/repos.json` [8].

3.2 Projects using Version Locking

The dataset described in Section 3.1 only considered projects that were recently active, to investigate how widespread the use of version locking is at the current moment. For research questions 2 and 3 we looked into how version locking is used, so we did not need to limit ourselves to recent projects. We therefore decided to use a different method to also include less recently updated projects, making sure we had as many datapoints as possible.

When a project uses version locking, Gradle generates a file called `gradle.lockfile`. We used the GitHub API to find all files called `gradle.lockfile` on GitHub. Unfortunately, due to restrictions on the Code Search endpoint of the GitHub API, we could only access the first 1000 results. To obtain even more results, we created two separate search queries: one for lock files in the root folder of the project and one for lock files in any other folder. Just like for research question 1, we filtered out projects with less than 10 stars, to not include any toy projects in our dataset. We stored the found lock files and their corresponding build files in a text file for later analysis. This resulted in a total of 327 different lock files, from 48 different repositories. One of these repositories (jjohannes/understanding-gradle) is a Gradle tutorial and therefore not a representable project. We thus removed it from our dataset, leaving us with 47 repositories.

The existence of a lock file alone does not necessarily mean that version locking was used. Version locking needs to be explicitly enabled somehow. The method described by Gradle is to enable it in the `build.gradle` file [1]. By manually looking through the projects in our dataset, we saw that 33 out of the 47 projects used this method and therefore definitely use version locking. For the other 14 projects it is unsure if they still use version locking, but since the lock file is there we know they at least used it at some point in time. We therefore decided it would still be interesting to include these projects in the dataset.

4 Adoptation of Version Locking (RQ1)

Firstly, we are looking into how much version locking is used in Gradle projects. Section 4.1 describes the methods used to investigate this, while Section 4.2 presents the results.

4.1 Methodology

The mining tool gave us a JSON file with 29,210 GitHub repositories (see Section 3.1). We wrote a script to filter out the projects using Gradle and see how many of those contained a lock file. Filtering out the projects using Gradle was made easy by the metadata for each repository provided by the mining tool [10]. It provides an attribute called "languages" for each repository, which lists all the languages used. If Gradle was in this list, we knew it was a project using Gradle.

To see if a repository contained a `gradle.lockfile`, we used the Code Search endpoint of the GitHub API. This endpoint has a rate limit of 10 requests per minute. To speed up the process we linked repositories together using the OR operator, so we could check multiple repositories in one API-request. According to the documentation,[1] we only 5 operators per request are allowed and a "Validation Failed" message will be returned for requests with more operators. However, while experimenting with the requests we found that this was not correct as requests with more than 5 operators still returned valid responses. The script therefore starts with 20 repositories per requests and breaks this number in half until the API accepts the request. This significantly sped up the process.

4.2 Results

After running the script on our dataset (see Section 3.1), we found that 12,553 of the 29,210 repositories use Gradle. Out of these repositories, only 43 (0.34%) contained a lock file. These results can be found in Table 1. The results also show that Gradle is quite often used in Java projects, with 42.98 percent of the sampled projects using it. We can conclude that version locking is not used often at all, despite it being a built-in feature in Gradle.

Metric	Count	Percentage
All repositories analyzed	29.210	100%
Gradle projects	12.553	42,98%
Projects using version locking	43	0,34% of Gradle projects

Table 1: Use of version locking in sampled projects

5 Version Ranges (RQ2)

Version locking was introduced in Gradle to ensure that projects do not break when using permissive version declarations, like version ranges [1]. We would therefore expect projects using version locking to also use version ranges. This section investigates if this is the case. In Section 5.1 the methodology is laid out. Section 5.2 presents the results of this investigation.

5.1 Methodology

To investigate how many of the projects using version locking also use version ranges, we wrote a script to look through the build files of projects using version locking. The script uses regular expressions to match version declarations that use one of the notations for version ranges (bracket notation, prefix notation or latest notation). The different repositories use different methods for declaring versions, so each of these methods was checked (see Section 2.1).

[1] https://docs.github.com/en/rest/search/search#limitations-on-query-length

The script keeps track of which dependencies use version ranges and how many of the range declarations are in the bracket, prefix or latest notation respectively. For a deeper insight into how permissive the ranges were, the "maximal variable part" of each dependency using version ranges is determined. We define the maximal variable part as the highest part of the version (major, minor or patch) that is allowed to change. For example, the maximal variable part of `2.+` is minor, because only the major part is pinned. For `[1.3, 2.5]` the maximal variable part is major, because the major version can be 1 or 2. For the latest notation, the maximal variable part is always major, as the version is not restricted in any way.

5.2 Results

After running the script, we find that 16 (34.0%) out of the 47 repositories in our dataset use version ranges. If we restrict ourselves to the repositories for which we are sure they are using version locking, the number becomes 13 (39.3%) out of 33. When looking deeper into how these ranges are declared (see Table 2), we see that most of the repositories using ranges use the latest notation (68.8%), which is the most permissive range declaration strategy.

Repository	Brackets	Prefix	Latest
ajoberstar/gradle-git-publish	1	0	0
ajoberstar/gradle-stutter	1	0	0
bwaldvogel/base91	0	0	4
bwaldvogel/liblinear-java	0	0	10
bwaldvogel/log4j-systemd-journal-appender	0	0	5
bwaldvogel/mongo-java-server	0	0	12
cronn/cucumber-junit5-example	0	0	5
cronn/jira-sync	0	0	4
cronn/reflection-util	0	0	15
cronn/ssh-proxy	0	0	13
cronn/validation-file-assertions	0	0	6
DataDog/dd-trace-java	9	237	0
datastax/fallout	0	21	0
jonatan-ivanov/resourceater	0	3	3
jonatan-ivanov/teahouse	0	1	9
MovingBlocks/TerasologyLauncher	1	0	0
Total	**12**	**262**	**86**

Table 2: Frequency of different notations for version ranges

Looking into the maximal variable part of each detected version range (see Table 3), we see that most repositories have version ranges that are not pinned on any part of the version and can therefore change in every part of the version (major, minor, patch). This is to be expected from the results above as the latest notation always has major as its maximal variable part. 5 repositories have dependencies with minor as its maximal variable part and only 3 repositories have dependencies with patch as its maximal variable part.

Repository	Major	Minor	Patch
ajoberstar/gradle-git-publish	0	1	0
ajoberstar/gradle-stutter	0	0	1
bwaldvogel/base91	4	0	0
bwaldvogel/liblinear-java	10	0	0
bwaldvogel/log4j-systemd-journal-appender	5	0	0
bwaldvogel/mongo-java-server	12	0	0
cronn/cucumber-junit5-example	5	0	0
cronn/jira-sync	4	0	0
cronn/reflection-util	15	0	0
cronn/ssh-proxy	13	0	0
cronn/validation-file-assertions	6	0	0
DataDog/dd-trace-java	101	93	52
datastax/fallout	2	13	6
jonatan-ivanov/resourceater	3	3	0
jonatan-ivanov/teahouse	9	1	0
MovingBlocks/TerasologyLauncher	1	0	0
Total	**190**	**111**	**59**

Table 3: Maximal variable part of version range per repository

To answer our research question, we can put a lower and upper bound on what percentage of repositories using version locking also use version ranges. This is because for some repositories, we are uncertain if they actually use version locking (see Section 3.2). The lower bound is the case where all of the uncertain repositories that also use ranges, do not use version locking and the other uncertain repositories do. In this case 13 out of 44 repositories that use version locking, also use ranges. The upper bound is the case in which all of the uncertain repositories that use ranges, do use version locking and the other uncertain repositories do not. In that case 16 out of 36 repositories using locking also use ranges. This means that the percentage of repositories using version locking that also use version ranges is between 29.5 and 44.4 percent.

6 Effects of Version Locking (RQ3)

We now know more about how version locking is used, but unclear is what the effects exactly are. To investigate this, we conducted the research laid out in this section.

6.1 Methodology

Version locking can have positive and negative effects. A positive effect occurs when a newer version of a locked dependency is released that would break the project. If this project would not be using version locking, the build would fail. A negative effect occurs when a newer version of a locked dependency is released. This newer version could include security patches, that are now not adopted by the project. To find out how often these two effects occur, we investigated them separately.

Positive effects

To see how often the positive effect occurs, we cloned the repositories and tried to build each of them to see if the build succeeded before removing the lock files. Because many of the projects use different Java versions, we manually inspected the `build.gradle` files of every project and noted down the Java version. We then wrote a script that, for every repository, used the Java version we detected and then tried to build the project. Then we removed the lock files of the

projects for which the build succeeded and tried to build the project again to see if this would cause the build to fail.

Negative effects
For the negative effect, we again cloned the repositories. We then copied all the lock files to a separate folder and resolved the dependencies for each of the repositories. This overwrites the old lock files with the newly generated ones. We then wrote a script to compare the old lock files with these new ones. The script detects per dependency, what happened to it after re-resolving. The following results are possible:

- **No change:** the dependency version stays the same in the new lock file
- **Added:** the dependency did not appear in the old lock file, but does appear in the new one
- **Removed:** the dependency appeared in the old lock file, but no longer appears in the new one
- **Major:** the dependency has a greater major version in the new lock file than in the old one
- **Minor:** the dependency has a greater minor version in the new lock file than in the old one
- **Patch:** the dependency has a greater patch version in the new lock file than in the old one

Some of the repositories have multiple subdirectories each with their own `build.gradle` and `gradle.lockfile` files. Furthermore, in a single `build.gradle` file, the same dependency can be set to different versions for different configurations. Because of these two reasons, the same dependency can have multiple different versions throughout a single repository. This also means that different things can happen to a single dependency after re-resolving. For example, a dependency could be removed in one lock file of a repository and updated in another lock file of the same repository. We could count all dependencies throughout all lock files in a repository, but decided against this as some repositories use many lock files, while others only have one. The repositories with many lock files would then be counted many times, which would skew the results.

We therefore decided on ranking the different possible outcomes for a dependency based on importance and only counting the most important change per dependency per repository. We chose the following order, from least to most important: no change, added, patch, minor, major, removed. 'No change' and 'added' both have no risk of introducing negative effects of version locking. For dependencies that did not change, the latest possible version is already used. For added dependencies, the version was not locked before so these also use the latest possible version. Major, minor and patch updates could have security issues if the newer version patches a security risk. Removed dependencies are an interesting case, as these also could have security risks. There are 2 reasons for a dependency to be removed from the new lock file. (1) The dependency could no longer be used by the repository we are looking into or (2) the removed dependency was a transitive dependency. In the first case there is no security risk as the repository is not using that dependency anymore. In the second case, however, the removed dependency is used by a dependency that is locked to a version where it was still using it. If the removed dependency has a security risk, it would have been removed if the repository was not using version locking, but now it is still being used.

6.2 Results

For this research question we first look into the positive effect version locking can have: ensuring a build always succeeds, even when breaking changes are introduced in newer versions of dependencies. Then we look at the possible negative effects: a project is using an older version of a dependency that possibly has security issues which are fixed in newer versions.

Positive effects
Because of various reasons (e.g. missing dependencies or projects using private packages), we were not able to build some of the projects at all. 11 out of the 47 projects built successfully, so we can use these projects as a dataset for this first part of the analysis. After the lock files were removed from these projects, 9 projects (81.8%) still built successfully and 2 projects (18.2%) fail the build. This means that version locking prevented the build to fail for 2 out of 11 projects. For these 2 projects, the reason for the unsuccessful build lies in the tests. We investigated these 2 projects further to understand the reason for the build to fail.

jonatan-ivanov/resourceater
This project failed because of a single test failing. The test that failed is the following:

```
@SpringBootTest(webEnvironment = RANDOM_PORT)
class ResourceaterApplicationTests {
    @Test void contextLoads() {}
}
```

Understanding exactly what happens here goes beyond the scope of this paper, but in short, this project uses Spring Boot to create a web-service.[2] The test automates testing if the web-service can start.[3] If it fails, the web-service for some reason cannot start, which is what happens here. If we look into the error message of this test, we see:

```
'Spring Boot [3.5.0] is not compatible with
this Spring Cloud release train', action =
'Change Spring Boot version to one of the
following versions [3.4.x]
```

If we look into the lock file of this repository, we indeed see that all Spring Boot dependencies were locked to version 3.4.2. After the lock file was removed, the newest version (3.5.0) was used, which according to the error message is not compatible with other dependencies.

bwaldvogel/log4j-systemd-journal-appender
This project failed because of a compilation error in at least one of the test files. Looking at the error message that Gradle gave us, we see:

[2] https://spring.io/projects/spring-boot
[3] https://spring.io/guides/gs/testing-web

```
Execution failed for task ':compileTestJava'.
> Compilation failed; see the compiler output
below.
```

In the compiler output we find:

```
import static org.mockito.Mockito.*;
                        ^
bad class file:
  class file has wrong version 55.0,
    should be 52.0
  Please remove or make sure it appears
    in the correct subdirectory of the
    classpath.
```

A class file is the compiled version of a Java file. Version 55.0 is equivalent to Java 11 and version 52.0 to Java 8 [11]. The test file is using Mockito, a Java framework used for testing. Looking at the README of Mockito's GitHub repository, we see that Mockito uses Java 11 for major version 5, while using Java 8 for major versions 3 and 4. If we look at the lock file, we see that version 4.2.0 of Mockito is locked. After we removed the lock file, Gradle resolved the newest version, version 5.18.0. Since this version uses Java 11 and the project uses Java 8, we get a compilation error.

Negative effects

As with building the projects, resolving the dependencies does not succeed for all the projects. It succeeds for 29 out of the 47 projects, which means we can use those as a dataset for this part of the research. After re-resolving the dependencies, we found that the lock files stayed the same for 16 out of the 29 repositories (55.2%). This means that for these repositories, all dependencies were locked to the version that were currently still the most recent version. Table 4 shows what happened to each of the dependencies for the other 13 repositories. What stands out are the results from "DataDog/vulnerable-java-application" and "filibuster-testing/filibuster-java-instrumentation". These repositories have many dependencies that no longer show up in the lock file after re-resolving the dependencies, while the other repositories have none or only a few. Because we are uncertain for either of these repositories if they actually use version locking (see Section 3.2) and they obviously show very different behavior, we removed them from the dataset. This left us with 11 out of 27 repositories (41.4%) for which the lock file changed after re-resolving the dependencies. Figure 1 shows what happened to each of the dependencies after re-resolving them. We see that the locked version for most of the dependencies (89.1%) stays the same. 26 (1.0%) of the dependencies got added, meaning they did not appear in the original lock file. For these two categories (90.2%) there is no negative effect; the dependencies that did not change are locked to the most recent version and the added dependencies are not locked at all, meaning they will also use the most recent version. For the rest of the dependencies (9.8%) negative effects could occur if the updates include security patches. We can therefore say the dependencies negatively affected by version locking are at most 9.8 percent.

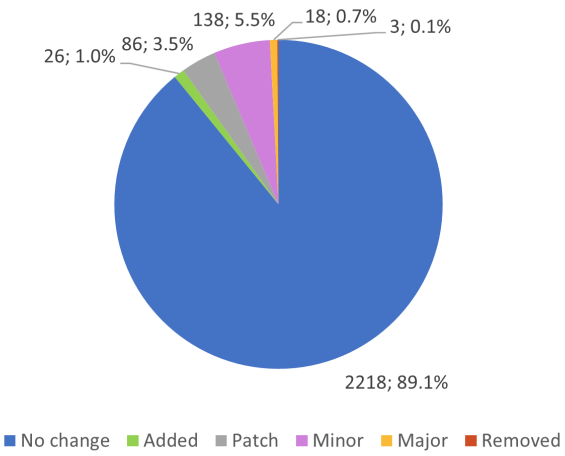

Figure 1: Result of re-resolving dependencies per dependency

7 Responsible Research

7.1 Reproducibility

Reproducibility is important for scientific research as it allows for verification of the results. Therefore, we have paid attention writing our methodology sections to ensure others can replicate our research and get the same results. All scripts used in our research our publicly available in our GitHub repository [8]. One threat to reproducibility could be the fact that the Code Search endpoint of the GitHub API, used to create the dataset in Section 3.2, does not produce the same response to the same request. We have countered this risk by providing the output files of this script in our GitHub repository. This means the same dataset can be used without using the script to create it.

Another threat could be the fact that our dataset includes active repositories. As the methodology for RQ3 includes cloning the repositories, this could yield different results at a later date if the build files or lock files have been changed. We cloned all repositories in our dataset at June 17th, 2025. Therefore, it is necessary to use the latest commit at that date for every repository, when replicating this research. When doing this, the research should yield the same results.

7.2 Integrity

In this research we maintained integrity by reporting all findings in an objective manner, without manipulation. When data points were removed from the tables or figures we mentioned this and argued why we believed this was reasonable. Furthermore, we presented and discussed the limitations of our research in Section 8.1.

7.3 Use of LLMs

In writing this paper we used Large Language Models only to generate tables from data and for inspiration. The generated tables were thoroughly checked for errors to make sure all presented results are correct. No output was directly copied and only general ideas, such as the structure of the paper, were used.

Repository	Removed	Major	Minor	Patch	Added	No change
ajoberstar/gradle-stutter	0	0	0	12	3	3
bwaldvogel/liblinear-java	0	1	12	8	0	14
bwaldvogel/log4j-systemd-journal-appender	0	1	6	0	0	2
bwaldvogel/mongo-java-server	0	0	32	18	4	9
cronn/reflection-util	0	0	7	1	0	22
cronn/ssh-proxy	0	0	7	0	0	24
cronn/validation-file-assertions	0	0	14	0	0	3
DataDog/vulnerable-java-application	68	0	0	0	0	0
filibuster-testing/filibuster-java-instrumentation	227	0	0	0	0	24
google/nomulus	0	0	3	4	0	475
gwtproject/gwt-http	0	9	0	1	7	122
gwtproject/gwt-places	2	5	0	1	8	124
jonatan-ivanov/resourceater	1	2	57	41	4	35

Table 4: Result of re-resolving dependencies per repository

8 Discussion

The results of this paper show that version locking is not widely adopted in projects using Gradle. Only 0.34% of the analyzed projects used version locking. This confirms the results from Gamage et al. [7], who found a percentage of 0.3 to 0.9. This low adoption could be due to a lack of awareness on the part of developers or a preference for up-to-date dependencies, to ensure that bugs and security issues in dependencies are fixed as soon as possible.

Furthermore, we found that version ranges are used in 29.5 to 44.4 percent of repositories using version locking. This is surprisingly little, as using version locking while not using version ranges has little effect. If ranges are not used, all direct dependencies are pinned to a specific version, meaning only transitive dependencies benefit from the version locking. This again could indicate a lack of awareness among developers about the use of version locking. The repositories that do use version ranges overall use very permissive ranges, with almost all of them (14/16) having dependencies with a maximal variable part of major, meaning they are not restricted to a single major version. This is more in line with our expectations as these repositories benefit the most from using version locking.

Looking into the effects that version locking has on the projects we found that 18.2% of the projects did not build after removing the lock files. This means that the vast majority of the projects (81.8%) did not benefit from locking at the moment of our research. It could be that these projects would fail without the lock files at a later point of time, especially when the lock files have recently been updated. It is still interesting to see that version locking was unnecessary at this point of time for most of the project. This could also explain why so few of all Gradle projects use version locking. Perhaps developers choose to pay attention to updated dependencies themselves when they build their projects and immediately update their projects when a dependency update causes their build to fail.

We also saw that for 41.4% of the projects, the lock files changed after re-resolving the dependencies, meaning they do not use the latest version allowed by the version declarations in the `build.gradle` file. As expected, the 2 repositories that failed the build after deleting the lock files are among these, as it were updated dependencies that made these builds fail. We concluded that at most 9.8% of all dependencies throughout the repositories are negatively effected by version locking. The locked versions of these dependencies may have security issues that are fixed in the newer versions. Future research could investigate how many of these actually have newer versions with security patches. Due to time constraints this was not included in this research.

8.1 Threats to Validity

Uncertainty about use of locking

For 14 out of the 47 projects from the dataset described in Section 3.2, it was unsure if they use version locking or not. A lock file was present, but locking was not activated in the `build.gradle` file. This is a potential threat to internal validity. We have therefore looked into some of these repositories to see what was happening. For some of them we were able to identify the commit where the activation of locking was removed from the `build.gradle` file. For example in `DataDog/dd-trace-java`, the `dependencyLocking` block (see Section 2.3) was removed in commit `86a24ac` and moved to a separate plugin. For other dependencies (like `ajoberstar/gradle-git-publish`), we were not able to identify the activation of version locking anywhere in the repository, not even in previous versions. However, the lock file was still updated multiple times throughout the history of the repository. This could be due to the developers updating the lock file locally and maybe even using locking when building the project, but for some reason choosing not to include this on GitHub. Due to these uncertainties about whether locking was (still) being used in these 14 repositories, we were only able to give a lower and upper bound for RQ2. We still decided to include these projects as they had either used version locking in the past, or were still using it locally. We believe that these repositories therefore are still interesting data points.

Selection bias

Another threat to internal validity is the fact that we could not use the full dataset for RQ3. Because of the different setups of the different repositories, not all projects could be built on our

machine. Similarly, resolving the dependencies also failed for some of the projects. This introduces a risk of selection bias as only the repositories that we managed to build or resolve were used in that part of our research. However, because the reasons for the build to fail were very diverse, we assume that there is no correlation between these projects. We therefore believe that the remaining projects still accurately represent the real world.

Implementation errors

Most of the results in this research were obtained by means of Python scripts that we wrote. We can never be entirely sure these scripts are without errors; however, we have tried to mitigate this risk by manually testing the scripts. For example, for the script that detects the use of version ranges in a `build.gradle` file, we randomly selected some of these files and checked if there were no false positives or negatives were detected by the script. Furthermore, all scripts used are publicly available in our GitHub repository [8].

8.2 Future Work

While our research provides some interesting insights into the use and effect of version locking, there are several areas that could be investigated further. Our research shows that version locking is not often used in Gradle projects, but it remains unclear why. Qualitative research involving interviews with developers could provide interesting insights into why version locking is so rarely used.

Furthermore, our research shows that at most 9.8% of dependencies are negatively affected by version locking, as they might have newer versions introducing security patches. Future research could investigate these dependencies and see if a security patch was introduced between the locked version and the newest version. This research could give more insights into the security risks introduced by the use of outdated dependencies in projects using version locking.

9 Conclusion

In this paper, we have investigated the use of version locking in Gradle projects. Version locking is a dependency management tool designed to ensure reproducible builds when using permissive dependency declaration strategies [1]. We analyzed over 12,000 Gradle-based Java projects and found that version locking is rarely used, with only 0.34% of the sampled projects containing a lock file.

We also looked into how often version ranges were used in projects using version locking. Only between 29.5% and 44.4% of the projects using version locking used version ranges. This is surprising as version locking is most useful when using version ranges. Without version ranges, dependencies are pinned to a specific version in their declaration so only transitive dependencies benefit from the version locking. This suggests that the optimal use of version locking might not be fully understood by developers.

To investigate the effects of version locking, we looked into how many of the projects benefited from using version locking by analyzing for how many of them the build would fail if version locking was not used. This was the case for 18.2% of the projects. On the other hand, up to 9.8% of the dependencies used in projects with version locking could be negatively affected by version locking, as there was a newer version available, possibly introducing security fixes, which was not used because the dependency was locked to an older version.

Seeing this low adoption rate, we draw the conclusion that there may be a need for better education on the benefits of version locking to developers. Future work could investigate the security implications further, which might also convince more developers to adopt it, as little information about these risks are currently known.

References

[1] Gradle, "Gradle Documentation." https://docs.gradle.org/current/userguide/dependency_locking.html, 6 2025.

[2] T. Preston-Werner, "Semantic Versioning 2.0.0." https://semver.org/.

[3] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the maven repository," *Journal of Systems and Software*, vol. 129, pp. 140–158, 7 2017.

[4] M. Keshani, S. Vos, and S. Proksch, "On the relation of method popularity to breaking changes in the maven ecosystem," *Journal of Systems and Software*, vol. 203, p. 111738, 9 2023.

[5] D. Venturini, F. R. Cogo, I. Polato, M. A. Gerosa, and I. S. Wiese, "I depended on you and you broke me: An empirical study of manifesting breaking changes in client packages," *ACM Transactions on Software Engineering and Methodology*, vol. 32, 5 2023.

[6] I. Pashchenko, D. L. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications," *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 1513 – 1531, 10 2020.

[7] Y. Gamage, D. Tiwari, M. Monperrus, and B. Baudry, "The design space of lockfiles across package managers," 2025.

[8] J. Boerop, "Scripts for research project." https://github.com/JBO393/CSE3000, 2025.

[9] J. Van Zyl and V. Siveton, "Maven Getting Started Guide – Maven." https://maven.apache.org/guides/getting-started/#What_is_a_SNAPSHOT_version.3F, 11 2006.

[10] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for MSR studies," in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, pp. 560–564, IEEE, 2021.

[11] M. Hoffmann, "Class File versions." https://javaalmanac.io/bytecode/versions/, 2018.