Pinning is Sinning: Towards Upgrading Maven Dependencies using Crowdsourced Tests

VASUDEV VIKRAM, Carnegie Mellon University, USA
YUVRAJ AGARWAL, Carnegie Mellon University, USA
ROHAN PADHYE, Carnegie Mellon University, USA

Library dependencies in software ecosystems play a crucial role in the development of software. As newer releases of these libraries are published, developers may opt to *pin* their dependencies to a particular version rather than upgrading to more recent ones. While pinning may have benefits in ensuring reproducible builds and avoiding breaking changes, it bears larger risks in using outdated dependencies that may contain bugs and security vulnerabilities. To understand the frequency and consequences of dependency pinning, we conduct an empirical study to show that over 60% of consumers of popular Maven libraries pin their dependencies to outdated versions, some over a year old. Furthermore, these pinned versions often miss out on security fixes; we find that upgrading dependencies to the latest minor or patch version is **3.45x** as likely to reduce security vulnerabilities rather than introduce new ones.

Consumers, however, may lack the confidence in performing an upgrade due to the possibility of introducing a breaking change. Thus, we propose Unpin, a novel tool that computes a confidence score for a dependency upgrade by leveraging crowdsourced tests of peer projects and simulating the upgrade for them. It can provide 35–100% more coverage of a dependency using only 1–5 additional test suites, compared that of a single consumer test suite. Our evaluation on real-world pins to the top 500 popular libraries in Maven shows that Unpin (with a minimum confidence score of 5) can provide confidence to over 3,000 consumers to safely perform an upgrade that reduces security vulnerabilities.

ACM Reference Format:
Vasudev Vikram, Yuvraj Agarwal, and Rohan Padhye. 2018. Pinning is Sinning: Towards Upgrading Maven Dependencies using Crowdsourced Tests. 1, 1 (October 2018), 20 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Modern software heavily relies on third-party libraries. Usage of these libraries can reduce software development time and cost by reusing existing functionality of software [1, 2]. This process has been integrated into many software ecosystems—such as Apache Maven for Java, NPM for JavaScript, and PIP for Python—for which building and installing library dependencies is a natural step for the software developer. The Maven Central Repository demonstrates the popularity of this practice for Java applications, with an index containing over 10 million Java packages [3]. An example of the dependency network of the Maven `gemini` library is shown in Figure 1, showing many dependencies than can span multiple edges.

While the dependence on third-party libraries assists the development of new software applications, managing these dependencies can be challenging. New releases of dependencies are

Authors' addresses: Vasudev Vikram, Carnegie Mellon University, Pittsburgh, PA, USA, vasumv@cmu.edu; Yuvraj Agarwal, Carnegie Mellon University, Pittsburgh, PA, USA, yuvraj@cs.cmu.edu; Rohan Padhye, Carnegie Mellon University, Pittsburgh, PA, USA, rohanpadhye@cmu.edu.

XXXX-XXXX/2018/10-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

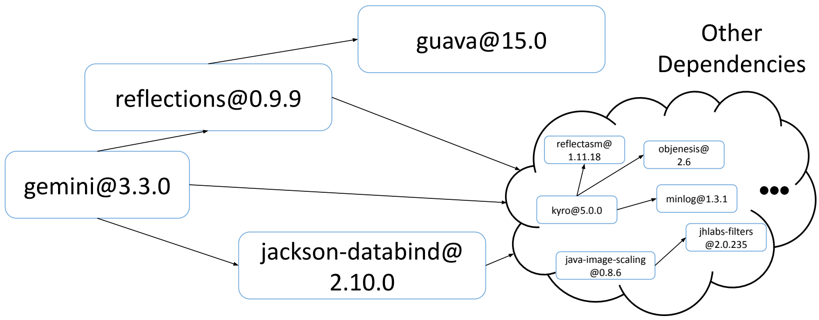

Fig. 1. Example dependency tree of the Maven library `gemini@3.3.0`. A directed arrow denotes a dependency. Each node consists of a library name and version. `gemini@3.3.0` contains a direct dependency to `jackson-databind@2.10.0` and an indirect dependency to `guava@15.0`.

constantly published to the ecosystem and developers must decide whether to upgrade them to a newer version. However, software bugs or unexpected behavior—referred to as breaking changes—can be introduced in these new versions [4–6]. Third-party library maintainers sometimes even *knowingly* deploy breaking changes due to the build up of technical debt and pressure to release new functionality [7].

Thus, upgrading a dependency can always be risky for consumers of these libraries. They must be wary of the possibility that their project might break or even that new security vulnerabilities are introduced [8]. This encourages developers to *pin* their dependencies to a specific version and avoiding performing dependency upgrades in their projects.

Dependency pinning may avoid this issue entirely and has certain benefits such as providing reproducible builds [9]; however, it bears a significant cost! New library versions often include new features, performance improvements, and crucial security patches. The high-profile 2017 Equifax data breach, in which a vulnerability in the open source Apache Struts library was exploited for leaking sensitive data of over 140 million consumers, demonstrates this drawback of pinning [10]. A patch for Apache Struts was available, but was not adopted by Equifax for over *two months*. Nowadays, tools like *Dependabot* and others [11–14] help warn developers about known security vulnerabilities in outdated dependencies, though this approach is reactive rather than proactive.

So, we ask: is dependency pinning actually worth it? We first conduct an empirical study on the Maven ecosystem to understand the how common the practice is and its broader security implications. We use the Open Source Insights dataset [15], recently published by Google, containing data about dependencies, consumers, and security vulnerabilities for over 569,000 Maven packages. We construct datasets from a targeted sample of the most popular Maven libraries from the Open Source Insights dataset and find that *over one-third* of these libraries contain at least one pin to their dependencies. Even further, *over 60%* of the consumers of the most popular libraries are pinned to outdated dependencies.

Given that dependency pinning is a fairly common practice in Maven, we next explore its security risks. Previous studies have shown that systems with outdated dependencies are four times likely to exhibit security vulnerabilities than those with fresh dependencies [16]. In our own historical analysis on pinned dependencies, we find that libraries would have been 3.45 times as likely to fix security vulnerabilities than introduce new ones had they *unpinned* their dependencies when publishing their library. This corresponds to over 22,000 consumers in our dataset that potentially could have fixed vulnerabilities (a majority of which having high or critical severity levels) had they been able to perform these upgrades. Hence, we conclude that *pinning is sinning*, as developers are far likelier to fix vulnerabilities by upgrading their outdated dependencies.

While the overall security benefit of unpinning is clear, we must still consider the aspect of evaluating whether performing a specific upgrade is safe. Our key insight is that the test suites of other consumers in the ecosystem can help validate the upgrade and provide more confidence to the developer. To this end, we propose *Unpin*, a tool that *crowdsources* test suites of peer consumers of the dependency to evaluate the safety of an upgrade. We specifically leverage the existence of *test-JARs* in the Maven ecosystem, which contain projects' compiled tests, in order to streamline the execution of consumer test suites. By executing these additional test suites against both the pinned version and upgraded version, we can characterize the impact of the upgrade on multiple projects. Unpin reports a *confidence score* of a particular upgrade determined by the number of consumer test suites that are able to successfully run when using the upgraded dependency version.

Is Unpin able to provide confidence to the consumers that could have performed vulnerability-fixing upgrades? In an evaluation of Unpin on our dataset of these upgrades, we first find that crowdsourcing just five consumer test suites is able to provide an average of almost 100% improvement in test coverage of a dependency over that of a single consumer. Unpin is able to provide a confidence score of at least *five* to over 3,000 consumers (15%) performing an upgrade that would fix security vulnerabilities.

In summary, this paper asks the following research questions:

RQ1: To what extent are libraries in the Maven ecosystem pinning dependencies?

RQ2: What is the security impact of pinning dependencies?

RQ3: How much can crowdsourced test suites improve coverage of the pinned dependency?

RQ4: Can crowdsourced test suites help validate vulnerability-fixing upgrades?

Our contributions are as follows:

(1) We conduct an empirical study on the Apache Maven ecosystem using the Open Source Insights dataset to determine the frequency and security impact of dependency pinning relating to the top 500 most-popular libraries.
(2) We present a tool *Unpin* that crowdsources consumer test suites to better characterize the safety of an upgrade across the network and provide confidence to developers when unpinning dependencies.
(3) We evaluate our tool on vulnerability-fixing upgrades in Maven libraries and find that Unpin is able to validate upgrades to over 3,000 consumers with a confidence score of 5.

2 BACKGROUND AND TERMINOLOGY

This section provides terminology that will be used in the paper and background on Maven, a software packaging ecosystem for Java.

2.1 Software Ecosystems

A software ecosystem is a collection of software libraries, each denoted by a name and a version number. We denote a library as $L@V$, where L refers to the library name and V refers to version. We define $\mathbb{L}$ as the set of all libraries in a particular software ecosystem, such as Maven for Java.

A library $L@V$ may contain a *direct dependency* to another library $L'@V'$, usually specified in a configuration file for the build system. Throughout this paper, we refer to a dependency as the specific package as pulled by the build system after dependency resolution. The dependency resolution process will resolve any wildcard versions or ranges specified in the configuration file and fetch one single version of the dependency. We refer to $L'@V'$ as a *direct dependency* and $L@V$ as a *direct consumer*. A shorthand notation for describing this direct dependency relation is $L@V \rightarrow L'@V'$. An example of a direct dependency relation can be seen in Figure 1 between `gemini@3.3.0` and `jackson-databind@2.10.0`. We define the entire dependency graph $\mathbb{G}$ as the set of all direct dependency relations (edges), and naturally define the functions *directDeps* and *directConsumers* to identify a direct dependency on D or a direct consumer C respectively as follows:

$$\begin{aligned} directDeps(L@V) &= \{D@V' \in \mathbb{L} \mid (L@V \rightarrow D@V') \in \mathbb{G}\} \\ directConsumers(L@V) &= \{C@V'' \in \mathbb{L} \mid (C@V'' \rightarrow L@V) \in \mathbb{G}\} \end{aligned}$$

A library dependency can also span multiple dependency edges, such as between `gemini@3.3.0` and `guava@15.0` in Figure 1. To account for these dependency relations, we define the function *allDeps* on $L@V$ to return the transitive closure of *directDeps* applied to $L@V$. We similarly define *allConsumers* as the transitive closure of *directConsumers*. These functions return the set of all dependencies and consumers of $L@V$, respectively, regardless of the number of edges. We additionally introduce the functions *indirectDeps* and *indirectConsumers* to return the sets of dependencies and consumers that are not direct.

A library has the option of *upgrading* a dependency from one version to a newer one. Continuing our example from Figure 1, the library `gemini@3.3.0` could upgrade `jackson-databind` from version `2.10.0` to `2.11.0`. We denote an *upgrade* as the pair $\langle D@V^{\alpha}, D@V^{\beta}\rangle$.

2.2 Semantic Versioning

When performing a dependency upgrade, it's crucial for consumers to understand the types of changes being introduced in a new dependency version and whether it is backwards compatible. One practice used in many software ecosystems is *semantic versioning* [17], which defines a set of rules for assigning version numbers to new releases of libraries. When using semantic versioning, a version V is structured into the format `major.minor.patch[-tag]`. For example, the dependency `jackson-databind` in Figure 1 has version `2.10.0`, where 2 is the major version, 1 is the minor version, and 10 is the patch version. For notational purposes, we define the functions *major*, *minor*, and *patch* to return the corresponding version numbers of a particular version V. This separation of version numbers also defines a total ordering between versions that compares major, minor, and patch versions numerically from left to right. We use this comparison logic throughout the paper when ordering versions (e.g. $V^{\beta} > V^{\alpha}$).

Semantic versioning is used to characterize the types of version upgrades in terms of backwards compatibility. Generally, version upgrades that include backwards *incompatible* changes increment the major version, whereas upgrades that do not break existing functionality are limited to minor or patch version increments. This allows library developers to notify consumers about the specific versions that introduce potential breaking changes, and consumers can choose which versions to adopt through a set of dependency constraints. Throughout this paper, we refer to minor and patch version upgrades as *semver-compatible*, as they should have the assurance of being backwards compatible.

Semantic versioning encourages consumers to perform semver-compatible upgrades on their dependencies since there should be no risk of introducing breaking changes. This can be as simple as specifying a version range for a dependency that freezes the major version, such as `[1.0.0, 2.0.0)`. However, semantic versioning is only a policy and is unenforceable throughout a software community; oftentimes new minor and patch versions may not respect the policy, resulting in unexpected breaking changes and upset consumers [18, 19]. These upgrades can even introduce accidental bugs or new security vulnerabilities, which may convince consumers to avoid semver-compatible upgrades entirely and decide to *pin* their dependencies to a single version.

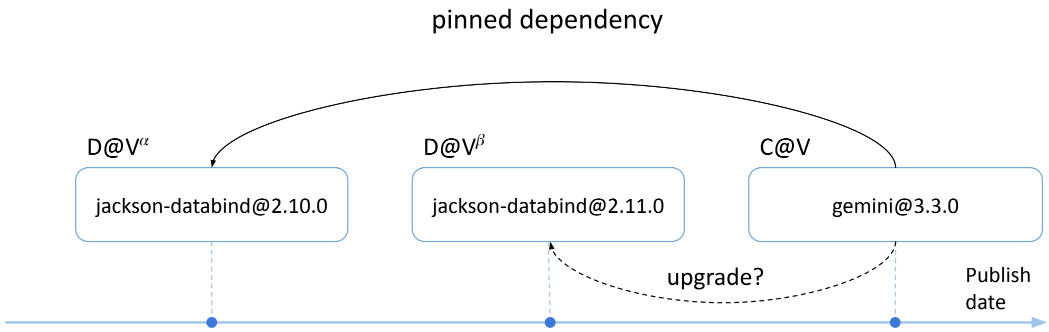

Fig. 2. Example of a direct pin between the consumer `gemini@3.3.0` and `jackson-databind@2.10.0`. Since `gemini@3.3.0` is a direct consumer of `jackson-databind@2.10.0` and a later version of the library `2.11.0` was published before the consumer, this is a pin.

2.3 Dependency Pinning

The practice of specifying a single version of a dependency rather than a range is referred to as *dependency pinning*. Figure 2 shows a pin in our previous example from the Maven library `gemini@3.3.0` to an outdated version of the `jackson-databind` library. When `gemini@3.3.0` was published, it contained a dependency to `jackson-databind@2.10.0` even though the later version `jackson-databind@2.11.0` was available. Although there was as an option to perform a semver-compatible upgrade, the consumer still kept the outdated version of the dependency.

We formally define a *pin* as follows: given a dependency graph $\mathbb{G}$, a pin is the tuple of three libraries $\langle C@V, D@V^{\alpha}, D@V^{\beta}\rangle \in \mathbb{L} \times \mathbb{L} \times \mathbb{L}$ for which the following conditions hold:

(1) $D@V^{\alpha} \in \mathit{allDeps}(C@V)$.
(2) $\mathit{publishTime}(V^{\alpha}) < \mathit{publishTime}(V^{\beta}) < \mathit{publishTime}(V)$.
(3) $(\mathit{major}(V^{\beta}) = \mathit{major}(V^{\alpha})) \wedge (V^{\beta} > V^{\alpha})$

The first condition specifies that a $D@V^{\alpha}$ is a dependency of consumer $C@V$. Next, the publish time of each of these libraries is compared: if the newer dependency version V^{β} was published before the consumer version V, then consumer $C@V$ is pinned to dependency $D@V^{\alpha}$, as it chose to use an outdated dependency version rather than performing the upgrade to V^{β}. The final condition incorporates semantic versioning guidelines and checks that the upgrade from V^{α} to V^{β} is a semver-compatible upgrade by ensuring major version equality and using the semantic versioning ordering. This filters out any major version upgrades due to their potential of introducing backwards incompatible changes.

We can further classify a pin as either *direct* or *indirect* depending on the nature of the dependency between $C@V$ and $D@V^{\alpha}$. $\langle C@V, D@V^{\alpha}, D@V^{\beta}\rangle$ is a direct pin if $D@V^{\alpha} \in \mathit{directDeps}(C@V)$ and a indirect pin if $D@V^{\alpha} \in \mathit{indirectDeps}(C@V)$. To *unpin* a direct pin, a consumer would simply need to update the version of the dependency to the newer version in the project configuration file. Unpinning indirect pins, on the other hand, requires the consumer to explicitly override the indirect dependency relation to $D@V^{\alpha}$ by introducing a new direct dependency relation to $D@V^{\beta}$.

Unpinning a dependency involves deciding to perform the upgrade from V^{α} (pinned version) to V^{β} (upgrade version) and is not necessarily a straightforward decision. Consumers may be apprehensive of incorporating changes that break their project or even introduce new security vulnerabilities. However, keeping the dependencies pinned has a risk of missing out on crucial patches for vulnerabilities that exist in the pinned version, usually fixed in minor and patch version

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
    ...
</project>
```

```
<dependencies>
    <dependency>
        <groupId>ant</groupId>
        <artifactId>ant-nodeps</artifactId>
        <version>1.6.5</version>
    </dependency>
</dependencies>
```

Fig. 3. Excerpt of the POM file for Apache `log4j:log4j@1.2.17` that lists a dependency on `ant:ant-nodeps@1.6.5`.

upgrades. Without a way of characterizing the impact of these upgrades beyond semantic versioning guidelines, developers must make a difficult decision when deciding to perform these dependency upgrades.

2.4 Apache Maven

For our empirical study and tool, we focused on the popular Apache Maven software ecosystem for Java projects. Maven provides support for building, managing, and deploying Java packages. Java files in Maven projects are usually organized into two directories: `src/main` and `src/test` files containing source and test code respectively.

Maven libraries can be uploaded as packages to the Maven Central Repository [3], which contains over 10 million indexed packages. Each package a binary JAR file of the compiled source Java classes (corresponding to the files in `src/main`) and a Project Object Model (POM) file. The POM file is an XML file that contains metadata, dependencies, and additional configurations of the project. An excerpt of a POM file for the Apache Log4j project can be seen in Figure 3. Libraries names are uniquely identified by the `<groupId>` and `<artifactId>`, and the version is specified under the `<version>` tag. Each dependency is listed under the `<dependencies>` tag by similarly specifying the `groupId`, `artifactId`, and `version`. A dependency version can be specified in the POM file with a single value or a version range (ref. Section 2.2). When the project builds, the Maven build system will parse the POM file, resolve a single version for each dependency, and fetch the corresponding JAR and POM files from the Maven Central Repository.

To run the unit and integration tests in the `src/test` directory of a Maven project, a developer can run the `mvn test` command in the project's source repository. For outsiders, replicating this process would require finding the source repository to clone, switching to the specific version of the library, and compiling the Java files in `src/main` and `src/test` before executing the tests. On the other hand, Maven projects have the option of uploading a *test-JAR* to the Central Maven Repository when deployed. A key insight is that a test-JAR can be used to *directly run* the unit and integration tests of a package without requiring access to the project's source repository. Test-JARs are a unique aspect of the Maven that provides access to many additional package tests in the ecosystem.

3 PINNING IN MAVEN

Using the Open Source Insights dataset published by Google [15], we conducted an analysis on a snapshot of the Maven ecosystem to measure the frequency and impact of pinning. We take a snapshot of the entire Maven dependency network on May 22, 2023 that includes dependencies and consumers (both direct and indirect) of ~567,000 Maven libraries. This snapshot contains 235,959,564 dependency edges, of which 45,997,607 (19.5%) are direct dependencies. Ignoring different versions, there are a total of 188,927 dependencies and 377,551 consumers.

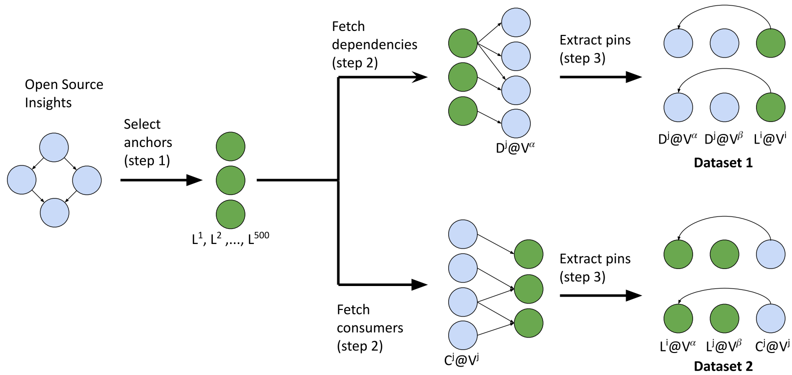

Fig. 4. Construction of pin datasets $\mathcal{D}_1$ and $\mathcal{D}_2$. A set of anchors of selected from the Open Source Insights dataset based on popularity (number of consumers). $\mathcal{D}_1$ is constructed by extracting pins from anchors to their dependencies, and $\mathcal{D}_2$ is constructed by extracting pins from the consumer of the anchors to the anchors themselves.

We chose to use this dataset because it includes the dependency versions that result from Maven's dependency resolution process rather than the syntax declared in the POM files of the projects. This provides resolution for version ranges or keywords in the POM file (e.g., `2.0+` or `LATEST`) and also solves version conflicts for duplicate indirect dependencies (i.e., diamonds in the dependency graph). By using the final resolved versions rather than declared versions, we can find *explicit* instances of pinning that occur in the ecosystem. To our knowledge, Open Source Insights is the most up-to-date dataset for Maven at the time of writing[1].

3.1 RQ1: Frequency of Pinning

In RQ1, we focus on how common the practice of dependency pinning is in the Maven ecosystem. Since the entire Maven ecosystem is too large to analyze in its entirety, we target our analysis to a sample of the Maven ecosystem relating to the top 500 most popular libraries (as defined by the number of consumers) due to their overall impact on the network. In particular, we analyze (1) pins of these most popular libraries to their dependencies and (2) pins of consumers to this set of the most popular libraries. We create two sub-questions for RQ1 accordingly:
RQ1.1: *Do the most popular Maven libraries pin dependencies?*
RQ1.2: *Do consumers pin to the most popular Maven libraries?*

For each sub-question, we construct a dataset of *pins* (as defined in Section 2.3) using the process shown in Figure 4. Each dataset uses the top 500 most popular libraries (referred to as *anchors*) as a starting point to find pins across the network. The anchors are created by selecting the library names (e.g., $L^1, L^2, \ldots, L^{500}$) with the highest number of consumers across all versions, as seen in Step 1 of Figure 4.

3.1.1 RQ1.1: Do the top 500 most popular Maven libraries pin dependencies? The dataset $\mathcal{D}_1$ consists of pins from the top 500 libraries to their dependencies. We first walk through an example with the Apache `avro` library to outline how $\mathcal{D}_1$ is constructed. The `avro` library is included as an anchor due to its high number of consumers. We first select the latest minor version of `avro` (`1.11.0`) as a recent version of this anchor. Next, we find all dependencies (direct and indirect) of `avro@1.11.0` and

[1]We originally used Libraries.io [20] for our dataset, which stores the dependency version as the syntax of the version listed in the POM files, but chose Open Source Insights due to its explicit versioning resolution and more up to date dataset.

Table 1. Pinning statistics for the top 500 most popular libraries to their dependencies, separated by direct and indirect relation. The number of consumers corresponds to the number of anchors that contain direct and indirect dependencies (many anchors have no dependencies to other libraries). Out of these consumers, 87 (34%) contain at least one direct pin and 73 (54%) contain at least one indirect pin. There are a total of 892 direct dependencies and 987 indirect dependencies across these consumers, of which 181 (20%) of them are direct pins and 364 (37%) are indirect pins.

	Anchors	**Anchors with ≥ 1 pin**	**Dependencies**	**Pins**
Direct	253	87	892	181
Indirect	134	73	987	364

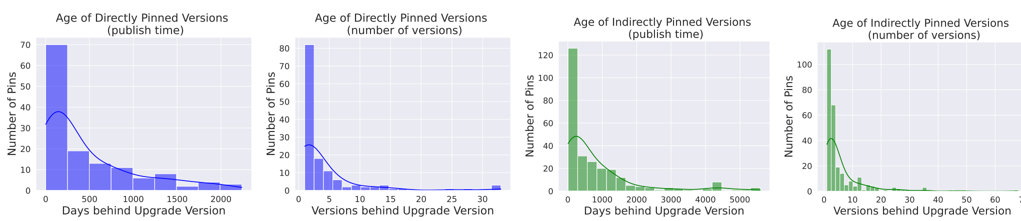

Fig. 5. Histograms showing the age of direct and indirect pinned dependencies for each Dataset $\mathcal{D}_1$. Direct pins are down in dark blue and indirect pins are show in light green. X-axis displays difference in publish time or version and Y-axis displays the number of pins. Values to the right represent pinned versions that are more outdated compared to the upgrade version.

check whether each one constitutes a pin. One such dependency is `jackson-databind@2.12.5`, for which there are multiple versions higher than `2.12.5` published before the date when `avro@1.11.0` was released. Since there may be many potential upgrade versions (e.g. `2.12.6`, `2.12.7`, etc.), we order all upgrade versions using semantic versioning and select the highest. Thus, the pin ⟨`avro@1.11.0`, `jackson-databind@2.12.5`, `jackson-databind@2.13.0`⟩ is added to $\mathcal{D}_1$.

Formally, we describe the process of constructing $\mathcal{D}_1$ as follows. We first use semantic versioning to select the latest minor version of each anchor and can denote these libraries $L^1@V^1, L^2@V^2, \ldots, L^{500}@V^{500}$. Next, we fetch all transitive dependencies (ref. Section 2.1) of all of the versioned anchors (Step 2 in Figure 4):

$$anchorDeps = \bigcup_{L^i@V^i} allDeps(L^i@V^j)\}$$

Finally, for each dependency $D^j@V^\alpha \in anchorDeps$, we query Open Source Insights to find the latest upgrade version of the dependency (V^β) that was published before the consumer $L^i@V^i$ (Step 3 of Figure 4). We then add the pin $\langle L^i@V^i, D^j@V^\alpha, D^j@V^\beta \rangle$ to set $\mathcal{D}_1$.

Table 1 provides statistics about the number of anchors, dependencies, and pins in $\mathcal{D}_1$. We first note that out of the 500 anchors, only 253 contain at least one direct dependency and 134 contain at least one indirect dependency. A large percentage (34.4%) of the anchors with direct dependencies contain at least 1 direct pin, and over half of the 134 anchors with indirect dependencies have at least 1 indirect pin. This is a significant portion of popular libraries that pin dependencies, which has downstream effects on the ecosystem: consumers that depend on these popular libraries are indirectly pinned to an outdated library!

For each of these pins, we would also like to measure how outdated the pinned version V^α is compared to the upgraded version available V^β. Figure 5 visualizes the difference in between the pinned version and the upgrade version in $\mathcal{D}_1$ by (1) publish time, and (2) number of versions released. Direct pins are shown in dark blue, and indirect pins are shown in light green. We observe that direct pins include a pinned version outdated by a median of 232 days and 2 versions behind the upgrade version; however, the majority of pinned versions are only 1 version behind. Indirect pins follow a similar trend with slightly more outdated pinned versions, having a median of 353 days and 3 versions behind the upgrade version.

Finding #1: A significant percentage of popular Maven libraries contain at least one pin to a dependency. However, the majority these dependencies are only moderately outdated by 1-2 versions.

3.1.2 RQ1.2: Do consumers pin to the top 500 most popular Maven libraries? We similarly construct dataset $\mathcal{D}_2$ to comprise of pins from other libraries to the anchors. Once again, we can walk through an example of extracting a pin for $\mathcal{D}_2$. We refer back to Figure 2 with the dependency from `gemini@3.3.0` to `jackson-databind@2.10.0`. As `jackson-databind` is one of our anchors, we would like to extract pins from consumers to its outdated versions. We begin by querying Open Source Insights to find all the consumers of `jackson-databind`, across all versions of the library. One such consumer is `gemini`—although there are many versions of this library, we select latest minor version (`3.3.0`) to find an up-to-date version. Since there are multiple versions of `jackson-databind` higher than version `2.10.0` published earlier than `gemini@3.3.0`, we select the highest one (`2.11.0`) and add the pin ⟨`gemini@3.3.0`, `jackson-databind@2.10.0`, `jackson-databind@2.11.0`⟩ to dataset $\mathcal{D}_2$.

The process of creating the entire dataset is formally described as follows: we first query the Open Source Insights network to find all consumers of the anchors libraries across all versions of each anchor, i.e.

$$anchorConsumers = \bigcup_{\substack{L^i@V^j \in \mathbb{L}\, \wedge \\ L^i \in anchors}} allConsumers(L^i@V^j)\}$$

We then query Open Source Insights to select the latest minor version of each consumer in *anchorConsumers*. For each consumer $C@V$, we find all of its dependencies to the anchor libraries and check whether any of them are pinned. Given a dependency to an anchor $L^i@V^\alpha$, we select the highest version V^β that was published before $C@V$ and add the corresponding pin to $\mathcal{D}_2$.

Table 2 shows the statistics of the number of consumers, dependencies, and upgrades in $\mathcal{D}_2$. Note that the total number of dependencies and consumers is much larger than $\mathcal{D}_1$. This is due to the selection of anchors; since the anchors are the top 500 most popular libraries by the count of the consumers who use them, it is natural that this dataset is much larger overall.

Interestingly, we find that *more than 60%* the direct consumers of the anchors contain at least 1 direct pin, and *over 80%* of the indirect consumers contain at least one indirect pin. Furthermore, we can see from Figure 6 that the dependency versions are outdated by a median of 370 days (7 versions) and 427 days (9 versions) for direct and indirect pins respectively. We see that pinning to the top 500 libraries is extremely common and features fairly outdated pinned versions! Note that there are a significantly smaller number of potential *upgrades* in $\mathcal{D}_2$ (as defined in Section 2.1) than there are pinning consumers, suggesting that many consumers share the same pins to the anchors.

Finding #2: Pinning to the most popular libraries in Maven is a very common practice, with over 60% of consumers containing at least one direct pin, and 80% containing at least one indirect pin. The pinned versions of these libraries are fairly outdated, about 7 versions behind the upgrade version for direct pins and 9 versions for indirect pins.

Table 2. Pinning statistics for consumers of the top 500 most popular libraries, categorized as either direct or indirect. Out of these consumers, 148,811 (61%) contain at least one direct pin and 184,281 (83%) contain at least one indirect pin. We see that many consumers share the same pins, as there are only 46,365 potential upgrades in the set of direct pins and 76,317 potential upgrades in the set of indirect pins.

	Consumers	Consumers with ≥ 1 pin	Dependencies	Potential Upgrades
Direct	244,819	148,811	717,705	46,365
Indirect	221,744	184,281	2,778,165	76,317

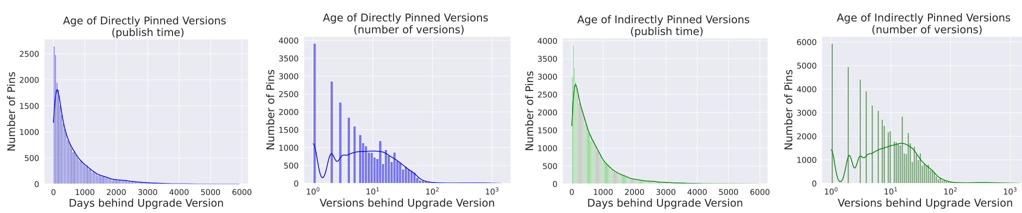

Fig. 6. Histograms showing the age of direct and indirect pinned dependencies for Dataset 2. Direct pins are down in dark blue and indirect pins are show in light green. X-axis displays age and Y-axis displays the number of pins. Log scale for X-axis is used for version plots.

3.2 RQ2: Security Impact of Unpinning

Older versions of libraries frequently contain known vulnerabilities that are patched in newer minor and patch releases. These security issues are tracked and disclosed publicly using Common Vulnerabilities and Exposures (CVEs) and other reporting mechanisms. The public Open Source Vulnerabilities (OSV) [21] database maintained by Google is a central database for CVEs and is used as a data source for the Open Source Insights dataset, which stored metadata about each vulnerability as an *advisory*. Each security advisory includes information about the packages and specific versions affected by the vulnerability.

From RQ1, we see that a very large percentage of consumers depend on an outdated version of the most popular libraries in the Maven ecosystem. While this provides a picture of how frequent dependency pinning occurs in the Maven ecosystem, we are interested in measuring the security impact of these pins: specifically, are developers avoiding introducing new security vulnerabilities into their dependencies by pinning, or they missing out on important security patches? Tools such as *dependabot* utilize these vulnerability databases to notify developers of vulnerable dependencies; however, this data has not been used to identify the historical security impact of pinned dependencies in Maven.

To perform this analysis, we compare the number of security vulnerabilities affecting the pinned version and upgrade version of the direct pins in dataset $\mathcal{D}_2$. Of the 46,365 potential upgrades (Table 2), we find that 40,462 result in no difference in vulnerabilities, 4,576 (9.9%) upgrades reduce the number of security vulnerabilities, and 1,327 (2.9%) introduce new ones. Thus, performing a semver-compatible upgrade of a pinned dependency in $\mathcal{D}_2$ is *3.45*× as likely to fix vulnerable dependencies than introduce new ones. Figure 7 displays the histogram of the differences in vulnerabilities between the versions, excluding the upgrades having no security impact for the sake of visualization. The majority of upgrades reduce the number of security vulnerabilities by 1, but certain upgrades can fix up to as many as 66 vulnerabilities! Across all of these upgrades, the number of vulnerabilities would be reduced by 20,825.

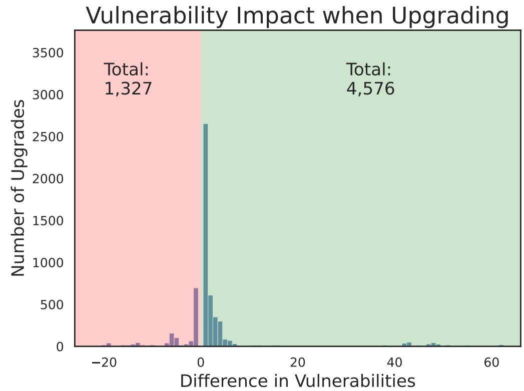

Fig. 7. Histogram visualizing the security advisory impact of upgrading directly pinned dependencies in $\mathcal{D}_2$. X-axis values in green include upgrades that reduce the number of vulnerabilities, whereas values in red increase the number of vulnerabilities (zero excluded for sake of visualization). In total, there are 4,576 upgrades that decrease vulnerabilities and 1,327 upgrades that increase vulnerabilities. Across all upgrades, the number of vulnerabilities reduce by 20,825.

➔ **Finding #3**: Performing a semver-compatible upgrade on a pinned version of a popular library is 3.45× as likely to reduce security vulnerabilities than introduce new ones. Thus, **pinning is sinning**.

4 SOLUTION APPROACH: UNPIN

In answering RQ1 and RQ2, we have identified that dependency pinning to the most popular libraries in Maven is fairly common and has high security risks. However, developers of these libraries may be cautious to perform these upgrades. To unpin a dependency, a consumer needs to be confident that the changes in the dependency upgrade are safe to introduce. One method would be to execute their test suites against the new version of the dependency. However, even if the tests pass, they may not be comprehensive enough to thoroughly test behaviors of the new dependency version. We address this concern by proposing a tool called *Unpin* that calculates a confidence score of a given upgrade by *crowdsourcing* test suites of other consumers of the pinned dependency. Our *key insight* is that consumer test suites can exercise a more thorough set of behaviors of the dependency; if multiple consumers' tests pass on both the pinned and upgraded version, a developer can more confidently unpin their dependency.

Unpin takes an upgrade $(D@V^{\alpha}, D@V^{\beta})$ and a minimum confidence setting $\mathcal{K}$ as input and validates the safety of that upgrade. The tool follows the procedure outlined in Figure 8:

(1) Query Open Source Insights to find $directConsumers(D@V^{\alpha})$.
(2) Pull the consumer test-JARs from the Maven Central Repository for each $C@V \in directConsumers$. Note that not all consumers have published test-JARs; thus, we construct a set $testableConsumers = \{C@V \in directConsumers(D@V^{\alpha}) \mid testJarExists(C@V)\}$
(3) For each consumer $C@V \in testableConsumers$, execute the tests when using $D@V^{\alpha}$ and $D@V^{\beta}$ as dependencies (see Section 4.1).
(4) Compare the test outcomes for each version and calculate a confidence for the upgrade $\langle D@V^{\alpha}, D@V^{\beta}\rangle$. If the confidence is at least $\mathcal{K}$, validate the upgrade (see Section 4.3).

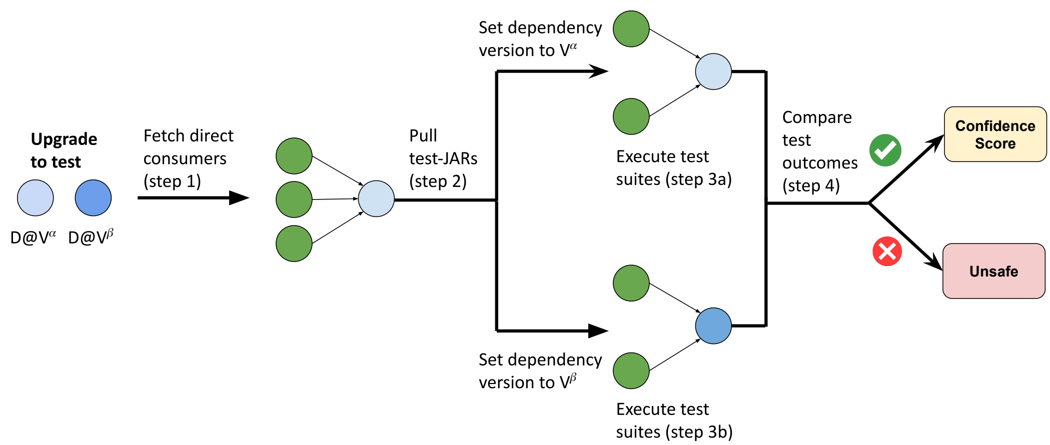

Fig. 8. Overview of Unpin. The direct consumers of $D@V^{\alpha}$ are fetched from Open Source Insights and their test suites are executed using test-JARs. Then, the version is set to V^{β} and the consumer test suites are executed on this version. Finally, the test outcomes are compared—either the upgrade is safe and Unpin returns a confidence equal to the number of test suites executed, or the upgrade is unsafe, returning zero confidence.

Steps (1) and (2) query the Open Source Insights dataset and the Maven Central Repository respectively to fetch test-JARs of the direct consumers of the pinned version. In the following sections, we go into detail to describe Steps (3) and (4).

4.1 Executing Crowdsourced Consumer Test Suites

One option to execute a consumer test suite is to download and build the source code of the repository and invoke the tests by running `mvn test`. Unfortunately, the source code for these consumers may not be publicly available. Additionally, resolving the specific version V in the repository can be a nontrivial task, as version naming conventions may differ between the source code and the Maven package.

The strategy we chose was to leverage the Maven Central Repository for *test-JARs* of the consumer, which contains compiled classes of the test files. Test-JARs are unique to the Maven ecosystem and provide a streamlined approach of fetching and executing project test suites. While test-JARs are optional to upload to the Maven Central Repository and do not exist for certain consumers, this approach still provides a straightforward method of crowdsourcing test suites. Among the consumers in $\mathcal{D}_2$ with direct pins, we found that about 12% of projects had uploaded test-JARs to the Maven Central Repository; while we would have liked this percentage to be higher, this is still a significant number of tests available for Unpin to use to test upgrades.

To walk through this process, we refer to our original example of a pinned dependency from `gemini@3.3.0` to `jackson-databind@2.10.0`. The consumer `gemini@3.3.0` would use Unpin to test the upgrade of `jackson-databind` from `2.10.0` to `2.11.0`. Unpin first finds all consumers of the pinned dependency `jackson-databind@2.10.0` and pulls all consumer test-JARs that are available on the Maven Central Repository. In the case that there are multiple consumers with the same library name, we select the highest version.

For each of the consumers, Unpin first executes each of the test suites against the pinned dependency version of `jackson-databind` (`2.10.0`). Some tests may produce non-deterministic outcomes due to *flakiness* [22, 23]. Unpin executes each test with $r = 5$ repetitions to account for this flakiness. Since the tests are executed directly from the test-JARs, it also is possible that tests may have errors or fail due to missing resources. We save the test outcomes produced by Maven of each of the consumer tests to a database.

Next, Unpin upgrades the dependency version of `jackson-databind` to `2.11.0` for each of the consumer test suites. Once again, the execution of the test suites are repeated five times, and the test outcomes are saved.

4.2 RQ3: Coverage Improvement of Crowdsourced Consumer Test Suites

A natural question, however, is whether using consumer test suites has any advantages in terms of exercising code, such as improved coverage, of the dependency? To characterize the coverage benefit of crowdsourced consumer test suites, we use the Jacoco library [24] to collect the coverage of the dependency classes only. Figure 9 shows the coverage improvement of consumer test suites for one pinned dependency `commons-io@2.4`—we found nine consumers of this dependency whose test JARs we could execute. From the figure, we see that the union of the coverage of these nine test suites provided over a 400% increase in coverage of `commons-io@2.4` than if we just executed a single consumer's test suite only (on average). To understand how coverage increases with the number of crowdsourced test suites, we calculate the union of the dependency-coverage for each value n below 9 by randomly sampling a subset of n consumer test suites without replacement (up to 50 times) and calculating the average.

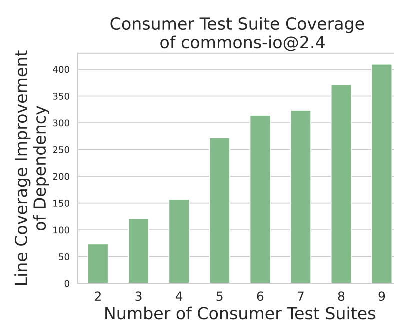

Fig. 9. Coverage improvement of consumer test suites for `commons-io@2.4`.

Generalizing this methodology, Figure 10 shows the average coverage improvement, across all popular libraries, with respect to the number of consumer test suites. With just a single additional consumer test suite, we can achieve an average of 40% additional coverage of the dependency; with four additional test suites, this number rises to almost 100%! The improvement saturates around 25 test suites, with about 300% improvement in coverage. Overall, we find that the crowdsourced test suites from Unpin are able to gain a significant coverage boost in the pinned dependency over a single consumer, thus providing more confidence in an upgrade.

4.3 Computing Confidence Score

We next explain how Unpin uses the outcomes from the consumer test suites to validate an upgrade. Based on the results of the crowdsourced test suites, Unpin calculates a *confidence* score for each upgrade. We walk through our example of upgrading `jackson-databind` from version `2.10.0` to `2.11.0`, with a minimum confidence setting of $\mathcal{K} = 5$. Unpin fetches and executes seven consumer test suites on the pinned version `2.10.0` and the upgrade version `2.11.0`. Tests that are flaky or fail in the pinned version are filtered out, and all remaining test outcomes are compared between versions. Each of the seven consumers *vote* on whether the upgrade is safe or unsafe. If all consumer tests pass on both dependency versions, then the consumer votes *safe*; otherwise, there exists a test that passes in the pinned version but fails in the upgrade version, indicating the presence of a breaking change. Since all seven consumers vote safe, the confidence returned by Unpin is seven. Since seven is higher than $\mathcal{K}$, Unpin validates this upgrade.

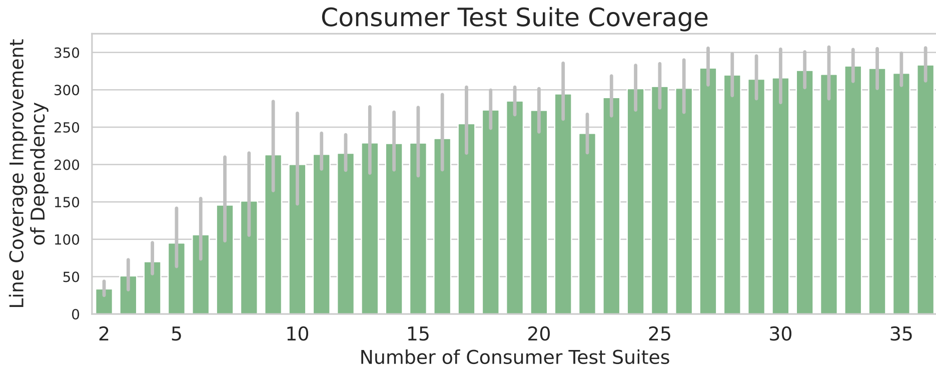

Fig. 10. Average coverage improvement achieved by Unpin over an average consumer test suite (higher is better). X-axis values include the number of crowdsourced consumer test suites, and Y-values show the geometric mean line-coverage improvement across all libraries. As low as four additional crowdsourced test suites can achieve almost 100% more line coverage than a single one.

More formally, we determine confidence as follows. We define *outcome* as a function that takes in a test method t, a consumer $C@V$, and a dependency $D@V$. From Section 4.1, each test has been executed with r repetitions.

$$outcome(t, C@V, D@V) = \begin{cases} pass & \text{if } r \text{ repetitions pass} \\ fail & \text{if } r \text{ repetitions fail or error} \\ flaky & \text{otherwise} \end{cases}$$

Each consumer provides a *vote* for whether the upgrade is safe or unsafe depending on the results of its test suite. If all passing tests with dependency version V^α also pass when the dependency version is upgraded to V^β, then the consumer vote is *safe*. If there is a test that consistently passes with V^α but always fails with V^β, then the consumer vote is *unsafe*—this condition indicates that the upgrade has broken some functionality. In all other cases (e.g., all tests were flaky or failed in V^α), the consumer vote is ignored.

$$vote(C@V, D@V^\alpha, D@V^\beta) = \begin{cases} safe & \text{if } \forall\, t \in consumerTests(C@V): \\ & \quad outcome(t, C@V, D@V^\alpha) = pass \implies \\ & \quad outcome(t, C@V, D@V^\beta) = pass \\ unsafe & \exists\, t \in consumerTests(C@V): \\ & \quad outcome(t, C@V, D@V^\alpha) = pass\, \land \\ & \quad outcome(t, C@V, D@V^\beta) = fail \\ ignore & \text{otherwise} \end{cases}$$

where $consumerTests(C@V)$ returns the set of all test methods in the test-JAR for $C@V$.

Finally, Unpin accumulates all votes of the consumers to calculate a *confidence* for the upgrade. If any consumers vote that the upgrade is unsafe, then the confidence is 0, since the upgrade appears to be a breaking change. Otherwise, the confidence is equal to the number of consumers that voted *safe*—higher is better. We formally define the confidence as follows:

Table 3. Unpin confidence on upgrades of direct pins $\mathcal{D}_2$ that reduce security vulnerabilities and the number of consumers affected. Out of the 4,576 upgrades, Unpin was able to crowdsource at least one test-JAR for 29% (upgrades with zero and positive confidence). Unpin returns a positive confidence for 9,194 (41%) of all consumers that could have performed these upgrades.

Confidence returned by Unpin	**Consumers**	**Upgrades**
Positive (upgrade is safe)	9,194 (41%)	850 (19%)
Zero (upgrade is unsafe)	3,134 (14%)	458 (10%)
Untested (upgrade had no test-JARs)	10,119 (45%)	3,268 (71%)
Total	22,447 (100%)	4,576 (100%)

$$
\mathit{confidence}(D@V^{\alpha}, D@V^{\beta}) = \begin{cases} 0 & \text{if } \exists\, C@V \in \mathit{testableConsumers}(D@V^{\alpha}) : \\ & \quad \mathit{vote}(C@V, D@V^{\alpha}, D@V^{\beta}) = \mathit{unsafe} \\ \displaystyle\sum_{\substack{C^i@V^i \in \\ \mathit{testableConsumers}(D@V^{\alpha})}} [\mathit{vote}(C^i@V^i, D@V^{\alpha}, D@V^{\beta}) = \mathit{safe}] & \text{otherwise} \end{cases}
$$

The confidence score calculated by Unpin reports the number of consumers that had consistent test results between dependency versions. In our example from earlier of the upgrade from `jackson-databind` from `2.10.0` to `2.11.0`, Unpin reports a confidence score of 7, since there were 7 consumer test suites executed. This score does not provide any guarantees about the safety of the upgrade—it is possible that the seven consumer test suites did not catch a breaking change. However, each additional consumer test suite provides more confidence, and the interpretation of the score is dependent on the preferences of the consumers performing the upgrade. The confidence scores reported Unpin will also increase with more testable consumers and more available test-JARs.

4.4 RQ4: Providing Confidence in Upgrades

A key question is whether Unpin can provide confidence to consumers of libraries to unpin one or more of their dependencies to upgrade them. We answer this RQ by running Unpin on the upgrades of direct pins in $\mathcal{D}_2$ that fix security vulnerabilities.

Table 3 reports the distribution of upgrades that had a positive and zero confidence returned by Unpin. About 29% of all upgrades were able to be tested with at least 1 test-JAR crowdsourced from the Maven Central Repository. Out of these tested upgrades, Unpin reported a positive confidence score for 850 (65%). This corresponds to 9,194 (41%) of all consumers that could have performed these upgrades.

We are also interested in how the minimum confidence setting $\mathcal{K}$ for Unpin relates to the number of consumers for which Unpin would validate the upgrade. Figure 11 visualizes these consumers against values of $\mathcal{K}$. The X-axis value of 1 is excluded for the sake of visualization and because we believe a minimum of 1 is too low. Overall, we find that with a minimum confidence setting of 5, over 3,000 (14%) of consumers would be able to validate their upgrade using Unpin. If the minimum confidence setting was set to 2, it would increase the number of consumers to almost *6,000*. This is a significant number of consumers that would be encouraged to upgrade their pinned dependencies with additional consumer test suites validating the upgrade. We believe this number can be increased even further with more Maven libraries adopting the practice of publishing their test-JARs.

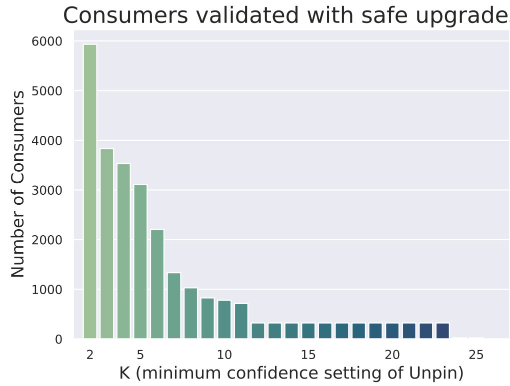

Fig. 11. Number of consumers with safe upgrades with respect to confidence score returned by Unpin. X-axis displays the minimum confidence score (1 is excluded for visualization), and Y-values are the number of consumers that would be able to unpin given the confidence value. Over 3,000 consumers could validate their upgrade using a minimum confidence setting of 5, and almost 6,000 using a minimum of 2.

5 DISCUSSION

In this section, we discuss our findings and their broader implications to practitioners and researchers.

Dependency pinning is common in the Maven ecosystem. From our analysis of dependency pinning in Maven, we find that pinning is fairly common for consumers of popular libraries, moreso than for popular libraries themselves. This is likely because popular libraries have more maintainers that can manage dependencies and keep them up to date. Additionally, it can be challenging for consumers to stay up to date with the frequent releases of popular libraries. While our analysis focuses on *explicit* instances of dependency pinning in the network, our findings are consistent with the studies evaluating the "freshness" of dependencies showing how developers are reluctant to upgrade their dependencies [16, 25–27].

Pinning is sinning. Our historical analysis of pinned dependencies to popular libraries shows that upgrading pinned version would have had a large security impact across the ecosystem. Although consumers may be inclined to stick to a consistent dependency version, they are far likelier to fix critical security vulnerabilities by keeping their dependencies up to date. This aligns with previous studies demonstrating correlations between outdated dependencies and vulnerabilities [28]. While we understand the benefits in fixing dependency versions, we hope this security implication encourages developers to adopt a more progressive strategy of upgrading dependencies.

Coverage of a dependency improves with crowdsourced test suites. It is challenging for a consumer to evaluate how their project will be affected by a dependency upgrade. While their own test suite may be able to catch certain issues, we see that *crowdsourcing* test suites from other consumers can provide a substantial boost in coverage. These test suites may be exercising different parts of the dependency, and a consumer may only care about a certain functionality that they use; nevertheless, we feel each additional test suite can only help in increasing confidence for an upgrade. Prior work has shown the potential for consumer tests [29–32] in achieving reasonable coverage and fault detection capabilities in dependencies.

Ecosystems should encourage developers to publicize executable test suites. Our tool Unpin leverages the published test-JARs in the Central Maven Repository. We believe this is a great practice to improve the overall testing infrastructure in the ecosystem and hope to see it more widely adopted by other libraries. In particular, the existence of test-JARs in the Central Maven Repository allows Unpin to streamline the automatic execution of these tests. This infrastructure is extremely valuable and hope to see it in other ecosystems beyond Maven/Java as well. Our approach of using external test suites to validate dependency changes is similar to how *monorepo* environments operate in large companies [33] in which tests from external modules are selected and run to validate code changes. Unpin applies this idea to the much broader open source world through the execution of consumer test suites, essentially providing something akin to a "monorepo for the masses".

6 THREATS TO VALIDITY

Threats to Construct Validity. The validation performed by Unpin on an upgrade is dependent on the consumer tests that are executed. If there is any noise or nondeterminism affecting the test outcome, then Unpin may improperly classify certain upgrades as safe or unsafe. This can arise from flakiness [22, 34, 35] in tests. We aim to mitigate this threat through repeated execution of the tests five times (Section 4.1) on both the pinned version and the upgrade version. Unpin only compares tests that produce a consistent passing or failing outcome across all repetitions, which should filter out a majority of flaky tests.

Threats to Internal Validity. Unpin's approach of crowdsourcing test suites and validating upgrades assumes that consumer test suites are a valuable source testing a dependency. Since library test suites are generally focused on testing functionality of the library and not the dependencies, it may be the case that consumer tests do not exercise much behavior of dependencies. Nevertheless, Unpin executes as many consumer test suites as are available in the Maven Central Repository. We hope that publishing test-JARs becomes a more widely adopted practice in Maven, as this would increase the overall coverage of the dependency.

Threats to External Validity. We specifically focused on the Maven ecosystem for our analysis, and we do not know if our conclusions about dependency pinning and its security implications will generalize to other ecosystems. Additionally, Unpin depends on a central repository of crowdsourced tests that can be automatically executed; this data may not always be available in other platforms.

7 RELATED WORK

7.1 Dependencies in Software Ecosystems

The challenge of evolving and maintaining software in ecosystems is a well-researched topic [36–39]. Bavota et al. [40] explore the Apache ecosystem and highlight the exponential growth in the number dependencies. They also found that application developers are reluctant to upgrade their dependencies due to the risk of API breaking changes. This issue is further quantified by Kula et al. (2015) [25], sampling 4.6K Github projects and finding that more than 80 percent of them have outdated Maven dependencies. Additional studies [41] validate this finding for other ecosystems such as NPM by measuring technical lag in dependencies. Dietrich et al. [27] demonstrate that 85.7% of Maven libraries specify a fixed version in dependencies—our definition of pinning is more precise as it compares the resolved version to the latest dependency version available at the time of publishing. Nevertheless, our analysis of our pin datasets confirms that outdated dependencies exist in a large percentage of libraries even in recent snapshots of the Maven ecosystem.

Prior work [42, 43] has also measured the impact of vulnerabilities in dependencies in the NPM ecosystem. Kula et al. (2018) [26] extend their work to study the extent to which developers upgrade their dependencies and the reasons behind their reluctancy [26]. In a survey of developers, they find that 69% claimed to be unaware of vulnerabilities in their dependencies. Automated dependency management bots like *Dependabot* [11] are able to address this issue by automatically notifying and creating pull requests for developers to upgrade their vulnerable dependencies. Analysis on Dependabot in practice shows that it does reduce technical lag in projects; however, its compatibility score does not reduce developer suspicion when performing upgrades [13]. Our approach can provide additional confidence through the execution of consumer test suites.

7.2 Detection of Breaking Changes

Prior research has studied [5, 7, 44] and developed numerous techniques for the detection of breaking changes [6, 45, 46] that can alert developers of unsafe upgrades.

Static Analysis Based Techniques. The majority of existing literature focuses primarily on detection of API changes between library versions. Raemaekers et al. [4] utlilize the tool `clirr` to detect API binary incompatibilities of Java code through static analysis. APIDiff is a tool developed by Brito et al. [45] that focuses on syntactic changes between Java library versions that classifies a code change as breaking or non-breaking. The more recent tool Sembid [47] locates breaking changes in Maven libraries by analyzing call chains and measuring semantic differences between versions.

Dynamic Analysis Based Techniques. Mostafa et al. [48] study the prevalence of *behavioral* backwards incompatibilities (BBIs) in consecutive versions of Java libraries. They find that 14 of the 15 subjects featured these types of breaking changes, with the majority of them undocumented. Prior work has also shown the effectiveness of using consumer tests to detect breaking changes and BBIs [29, 31, 47]. We highlight the main differences from our work: first, we provide a novel definition of explicit dependency pins and present a thorough empirical study on pinning in the Maven network, which is unique among related work. We also use a dataset that resolves dependency versions for old libraries at the time they were built; this is contrast to prior work that uses heuristics to resolves dependencies in older releases [31]. We focus on the security impact of pinning dependencies and validating upgrades from pins, which is unique among related work. Finally, we use crowdsourced tests from JARs published to the Maven central repository, and thus do not rely on identifying source code repositories like prior work [29–31].

8 CONCLUSION

In this work, we focused on the issue of dependency pinning in the Maven ecosystem. We conducted an analysis on a recent snapshot of the Maven ecosystem and identified that a significant portion of consumers are pinned to older versions of the most popular libraries. We also show that consumers are far more likely to fix existing security vulnerabilities than introduce new ones if they were to upgrade their outdated dependencies. To encourage developers to upgrade dependencies, we propose Unpin, a tool to execute crowdsourced consumer test suites in order to validate an upgrade. We find that Unpin is able to provide validation to over 19% of all consumers in our dataset performing upgrades that would have fixed known vulnerabilities. We argue that more libraries and package management platforms should adopt the practice of publishing executable test binaries which would allow further development of tools that leverage information about dependency usage via crowdsourced tests.

9 DATA AVAILABILITY

We have included evaluation data in the anonymized repository at: https://doi.org/10.5281/zenodo.8384971. This data contains dependency data for each of the datasets, coverage data for consumer test suites, and test outcome data from Unpin.

REFERENCES

[1] C. R. de Souza and D. F. Redmiles, "An empirical study of software developers' management of dependencies and changes," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 241–250.
[2] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, "Software dependencies, work dependencies, and their impact on failures," *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 864–878, 2009.
[3] "The maven central repository," https://mvnrepository.com/repos/central, accessed: 2022-11-21.
[4] S. Raemaekers, A. Van Deursen, and J. Visser, "Semantic versioning versus breaking changes: A study of the maven repository," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation.* IEEE, 2014, pp. 215–224.
[5] L. Xavier, A. Brito, A. Hora, and M. T. Valente, "Historical and impact analysis of api breaking changes: A large-scale study," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 2017, pp. 138–147.
[6] L. Ochoa, T. Degueule, J.-R. Falleri, and J. Vinju, "Breaking bad? semantic versioning and impact of breaking changes in maven central: An external and differentiated replication study," *Empirical Software Engineering*, vol. 27, no. 3, p. 61, 2022.
[7] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an api: cost negotiation and community values in three software ecosystems," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 109–120.
[8] S. E. Ponta, H. Plate, and A. Sabetta, "Detection, assessment and mitigation of vulnerabilities in open source dependencies," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3175–3215, 2020.
[9] S. Mukherjee, A. Almanza, and C. Rubio-González, "Fixing dependency errors for python build reproducibility," in *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*, 2021, pp. 439–451.
[10] M. Corporation, "CVE-2017-5638," https://www.cve.org/CVERecord?id=CVE-2017-5638, 2017. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2017-5638
[11] Github, "Dependabot," https://docs.github.com/en/code-security/dependabot/working-with-dependabot/automating-dependabot-with-github-actions#about-dependabot-and-github-actions, 2021, retrieved June 1, 2023.
[12] M. Alfadel, D. E. Costa, E. Shihab, and M. Mkhallalati, "On the use of dependabot security pull requests," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR).* IEEE, 2021, pp. 254–265.
[13] R. He, H. He, Y. Zhang, and M. Zhou, "Automating dependency updates in practice: An exploratory study on github dependabot," *IEEE Transactions on Software Engineering*, 2023.
[14] H. Mohayeji, A. Agaronian, E. Constantinou, N. Zannone, and A. Serebrenik, "Investigating the resolution of vulnerable dependencies with dependabot security updates," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR).* IEEE, 2023, pp. 234–246.
[15] Google, "Open Source Insights," https://deps.dev/, 2023, retrieved June 1, 2023.
[16] J. Cox, E. Bouwers, M. Van Eekelen, and J. Visser, "Measuring dependency freshness in software systems," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 109–118.
[17] T. Preston-Werner, "Semantic versioning 2.0.0," https://semver.org/spec/v2.0.0.html, 2013.
[18] A. Decan and T. Mens, "What do package dependencies tell us about semantic versioning?" *IEEE Transactions on Software Engineering*, vol. PP, pp. 1–1, 05 2019.
[19] 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*, 2016, pp. 351–361.
[20] Libraries.io, "Libraries.io Open Data," https://libraries.io/data, 2020, retrieved June 1, 2023.
[21] Google, "Open Source Vulnerabilities," https://osv.dev/, retrieved June 1, 2023.
[22] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 643–653.
[23] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 830–840.
[24] M. R. Hoffmann, B. Janiczak, and E. Mandrikov, "Eclemma-jacoco Java code coverage library," 2011.
[25] R. G. Kula, D. M. German, T. Ishio, and K. Inoue, "Trusting a library: A study of the latency to adopt the latest maven release," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER).* IEEE, 2015, pp. 520–524.
[26] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies? an empirical study on the impact of security advisories on library migration," *Empirical Software Engineering*, vol. 23, pp. 384–417, 2018.
[27] J. Dietrich, D. Pearce, J. Stringer, A. Tahir, and K. Blincoe, "Dependency versioning in the wild," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR).* IEEE, 2019, pp. 349–359.

[28] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vulnerable open source dependencies: Counting those that matter," in *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*, 2018, pp. 1–10.
[29] G. Mezzetti, A. Møller, and M. T. Torp, "Type regression testing to detect breaking changes in node. js libraries," in *32nd european conference on object-oriented programming (ECOOP 2018).* Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.
[30] L. Chen, F. Hassan, X. Wang, and L. Zhang, "Taming behavioral backward incompatibilities via cross-project testing and analysis," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 112–124.
[31] S. Mujahid, R. Abdalkareem, E. Shihab, and S. McIntosh, "Using others' tests to identify breaking updates," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 466–476.
[32] J. Hejderup and G. Gousios, "Can we trust tests to automate dependency updates? a case study of Java projects," *Journal of Systems and Software*, vol. 183, p. 111097, 2022.
[33] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP).* IEEE, 2017, pp. 233–242.
[34] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1471–1482.
[35] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–74, 2021.
[36] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance," *Journal of software maintenance and evolution: Research and Practice*, vol. 13, no. 1, pp. 3–30, 2001.
[37] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, "Challenges in software evolution," in *Eighth International Workshop on Principles of Software Evolution (IWPSE'05).* IEEE, 2005, pp. 13–22.
[38] K. Manikas and K. M. Hansen, "Software ecosystems–a systematic literature review," *Journal of Systems and Software*, vol. 86, no. 5, pp. 1294–1306, 2013.
[39] R. Cox, "Surviving software dependencies," *Communications of the ACM*, vol. 62, no. 9, pp. 36–43, 2019.
[40] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the apache community upgrades dependencies: an evolutionary study," *Empirical Software Engineering*, vol. 20, no. 5, pp. 1275–1317, 2015.
[41] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. González-Barahona, "An empirical analysis of technical lag in npm package dependencies," in *International Conference on Software Reuse.* Springer, 2018, pp. 95–110.
[42] 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 *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 672–684.
[43] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, 2019.
[44] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "When and how to make breaking changes: Policies and practices in 18 open source software ecosystems," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 4, pp. 1–56, 2021.
[45] A. Brito, L. Xavier, A. Hora, and M. T. Valente, "Apidiff: Detecting api breaking changes," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 2018, pp. 507–511.
[46] X. Du and J. Ma, "Aexpy: Detecting api breaking changes in python packages," in *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE).* IEEE, 2022, pp. 470–481.
[47] L. Zhang, C. Liu, Z. Xu, S. Chen, L. Fan, B. Chen, and Y. Liu, "Has my release disobeyed semantic versioning? static detection based on semantic differencing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.
[48] S. Mostafa, R. Rodriguez, and X. Wang, "Experience paper: a study on behavioral backward incompatibilities of Java software libraries," in *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis*, 2017, pp. 215–225.