Noname manuscript No.
(will be inserted by the editor)

Do Developers Update Their Library Dependencies?

An Empirical Study on the Impact of Security Advisories on Library Migration

Raula Gaikovina Kula · Daniel M. German · Ali Ouni · Takashi Ishio · Katsuro Inoue

Received: date / Accepted: date

Abstract Third-party library reuse has become common practice in contemporary software development, as it includes several benefits for developers. Library dependencies are constantly evolving, with newly added features and patches that fix bugs in older versions. To take full advantage of third-party reuse, developers should always keep up to date with the latest versions of their library dependencies. In this paper, we investigate the extent of which developers update their library dependencies. Specifically, we conducted an empirical study on library migration that covers over 4,600 GitHub software projects and 2,700 library dependencies. Results show that although many of these systems rely heavily on dependencies, 81.5% of the studied systems still keep their outdated dependencies. In the case of updating a vulnerable dependency, the study reveals that affected developers are not likely to respond to a security advisory. Surveying these developers, we find that 69% of the interviewees claimed to be unaware of their vulnerable dependencies. Moreover, developers are not likely to prioritize a library update, as it is perceived to be extra workload and responsibility. This study concludes that even though third-party reuse is common practice, updating a dependency is not as common for many developers.

Keywords software reuse, software maintenance, security vulnerabilities

Raula Gaikovina Kula[a b], Ali Ouni[a c], Takashi Ishio[a b] and Katsuro Inoue[a]
[a]Osaka University, Japan E-mail: {raula-k, ali, ishio, inoue}@ist.osaka-u.ac.jp
[b]Nara Institute of Science and Technology E-mail: {raula-k, ishio}@is.naist.jp
[c]UAE University, UAE E-mail: ouniali@gmail.com

Daniel M. German
University of Victoria, Canada
E-mail: dmg@uvic.ca

1 Introduction

In contemporary software development, developers often rely on third-party libraries to provide a specific functionality in their applications. In 2010, Sonatype reported that Maven Central[1] contained over 260,000 maven libraries[2]. As of November 2016, this collection of libraries rose to 1,669,639 unique Maven libraries[3], which is almost six times more than it was in 2010 and making it one of the largest hosting repositories of Open Source Software (OSS) libraries. Libraries aim to save both time and resources and reduce redundancy by taking advantage of existing quality implementations.

Many libraries are in constant evolution, releasing newer versions that fix defects, patch vulnerabilities and enhance features. In fact, Lehman [1996] states that software either *'undergoes continual changes or becomes progressively less useful'.* As software development transitions into the maintenance phase, a developer later becomes the maintainer and faces the following software maintenance dilemma: *'When should I update my current library dependencies?'* We define this dilemma of updating libraries as the *library migration process*, where there is movement from a specific library version towards a newer version of the same library, or to a different library altogether.

The decision to migrate a library can range from being rather trivial to extremely difficult. Typically, a developer evaluates the overall quality of the new release version, taking into account: (i) new features, (ii) compatibility compared to the current version, (iii) popular usage by other systems and (iv) documentation, support and longevity provided by the library. Conversely, a vulnerable dependency requires an immediate response from the developer. To mitigate any potential malicious attacks, developers are strongly recommended to migrate any vulnerable dependencies in their applications. In response to these vulnerable dependencies, *awareness mechanisms* such as the `Common Vulnerabilities and Exposures` (CVE)[4] advisories have emerged to raise developer awareness and trigger the migration from a vulnerable dependency to a safer replacement.

In this paper, we investigate the extent of how library migration is practiced in the real-world. Our goals are to investigate (1) whether or not library dependencies are being updated and (2) the level of developer awareness to library migration opportunities. Specifically, we performed a large-scale empirical study to track library migrations between an application client (defined as a system) and their dependent library provider (defined as a library). The study encompasses 4,659 projects, 8 case studies and a developer survey to draw the following conclusions:
(1) Library Migration in Practice: Although systems depend heavily on libraries, findings show that many of these systems rarely update their library

[1] one of the largest library hosting repositories at `http://search.maven.org/`
[2] Link at `http://goo.gl/SV9d68`
[3] statistics accessed Nov-26th-2016 at `https://search.maven.org/\#stats`
[4] `http://cve.mitre.org/cve/index.html`

dependencies. Developers are less likely to migrate their library dependencies, with up to 81.5% of systems keeping outdated dependencies.

(2) Developer Responsiveness to Awareness Mechanisms: Our findings indicates different patterns of library migration. We find many cases where developers prefer an older and popular dependency over a newer replacement. Importantly, the study depicts developers as being non responsive to security advisories. In a follow-up survey of affected developers, 69% of the interviewees claimed to be unaware of the vulnerability and who then promptly migrated away from the vulnerable dependency. Furthermore, developers cite (i) a lack of awareness in regard to library migration opportunities, (ii) the impact and priority of the dependency, and (iii) the assigned roles and responsibilities as deciding factors on whether or not they should migrate a library dependency.

Our main contributions are three-fold. Our first contribution is a study on library migration pertaining to developer responsiveness to existing awareness mechanisms (i.e., security advisories). Our second contribution is the modeling of library migration from system and library dimensions, with different metrics and visualizations such as the Library Migration Plot (LMP). Finally, we make available our dataset of 852,322 library dependency migrations. All our tools and data are publicly available from the paper's replication package at `https://raux.github.io/Impact-of-Security-Advisories-on-Library-Migrations/`.

1.1 Paper Organization

The rest of the paper is organized as follows. Section 2 describes the basic concepts of library migrations and awareness mechanisms. Section 3 motivates our research questions, while Section 4 describes our research methods to address them. The results and case studies of the empirical study are presented in Section 5 and Section 6. We then discuss implications of our results and the validity threats in Section 7, with Section 8 surveying related works. Finally, Section 9 concludes our paper.

2 Basic Concepts & Definitions

In this section, we introduce the library migration process and the related terminologies that will be used in the paper. Building on our previous work of trusting the latest versions of libraries [Kula et al., 2015] and visualizing the evolution of libraries [Kula et al., 2014], this paper is concerned with empirically tracking library migration and understanding the awareness mechanisms that trigger the migration process. We first present the library migration process in Section 2.1. Then later in Section 2.2, we introduce two common awareness mechanisms that are designed to trigger a library migration.

2.1 The Library Migration Process

We identify three generic steps performed by a developer during the library migration process:

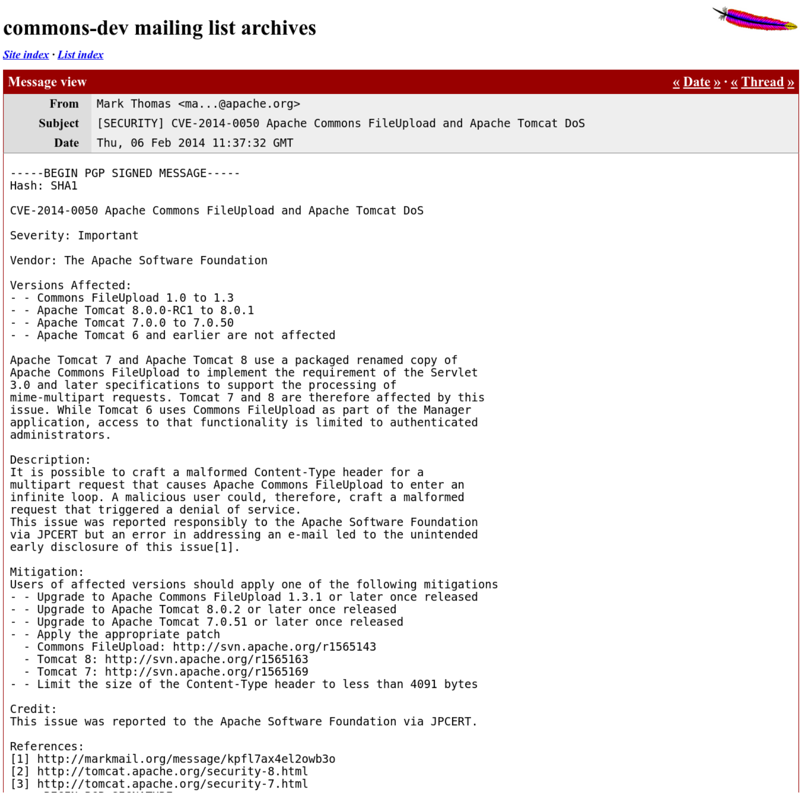
- *Step 1: Awareness of a Library Migration Opportunity*. Step 1 is triggered when a developer becomes aware of an opportunity to migrate a specific dependency. The awareness mechanism may be in the form of either a new release announcement by authors of the library or a security advisory. In order for a successful migration, a developer must also identify a suitable replacement for the current dependency. In the case of a vulnerable dependency, a developer must identify a safe (patched) library version as a viable replacement candidate.
- *Step 2: Migration Effort to Facilitate the Replacement Dependency*. Step 2 involves the efforts of a developer to ensure that the replacement dependency is successfully integrated into the system. Specifically, we define this *migration effort* as the amount of work and testing needed to facilitate the replacement dependency. This step may involve writing additional integration code and testing to make sure that the replacement library does not break current functionality, or affect other dependencies that co-exist within the system.
- *Step 3: Performing the Library Migration*. Step 3 ends the library migration process. Once the migration effort in Step 2 is completed, the prior dependency is abandoned and the replacement library successfully migrated into the system.

2.2 Library Migration Awareness Mechanisms

To trigger the library migration process, developers must first become aware of the necessity to migrate any of their dependencies. In this section, we discuss the two most common types of awareness mechanisms: (1) a new version release announcement and (2) a security advisory. Additionally, we present sources that may infer the migration effort required to migrate these dependencies.

(1) A New Release Announcement: The traditional method to raise awareness of a new release is through an announcement from the official homepage of the library. In detail, we can infer the migration effort required from the following two documented sources:

(i) *Change logs of releases* - The developer change logs provide hints on the migration effort needed to perform the migration. For instance, a new release to support the state-of-the-art environments (i.e., such as the Java Development Kit (JDK)) is more likely to require more migration effort, especially if the new version breaks many of Application Programming Interfaces (APIs). The change logs contains useful information, such as any API changes between releases, new features and fixes to bugs that exist in older versions.



commons-dev mailing list archives

Site index · List index

Message view « Date » · « Thread »

From	Mark Thomas <ma...@apache.org>
Subject	[SECURITY] CVE-2014-0050 Apache Commons FileUpload and Apache Tomcat DoS
Date	Thu, 06 Feb 2014 11:37:32 GMT

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

CVE-2014-0050 Apache Commons FileUpload and Apache Tomcat DoS

Severity: Important

Vendor: The Apache Software Foundation

Versions Affected:
- - Commons FileUpload 1.0 to 1.3
- - Apache Tomcat 8.0.0-RC1 to 8.0.1
- - Apache Tomcat 7.0.0 to 7.0.50
- - Apache Tomcat 6 and earlier are not affected

Apache Tomcat 7 and Apache Tomcat 8 use a packaged renamed copy of
Apache Commons FileUpload to implement the requirement of the Servlet
3.0 and later specifications to support the processing of
mime-multipart requests. Tomcat 7 and 8 are therefore affected by this
issue. While Tomcat 6 uses Commons FileUpload as part of the Manager
application, access to that functionality is limited to authenticated
administrators.

Description:
It is possible to craft a malformed Content-Type header for a
multipart request that causes Apache Commons FileUpload to enter an
infinite loop. A malicious user could, therefore, craft a malformed
request that triggered a denial of service.
This issue was reported responsibly to the Apache Software Foundation
via JPCERT but an error in addressing an e-mail led to the unintended
early disclosure of this issue[1].

Mitigation:
Users of affected versions should apply one of the following mitigations
- - Upgrade to Apache Commons FileUpload 1.3.1 or later once released
- - Upgrade to Apache Tomcat 8.0.2 or later once released
- - Upgrade to Apache Tomcat 7.0.51 or later once released
- - Apply the appropriate patch
  - Commons FileUpload: http://svn.apache.org/r1565143
  - Tomcat 8: http://svn.apache.org/r1565163
  - Tomcat 7: http://svn.apache.org/r1565169
- - Limit the size of the Content-Type header to less than 4091 bytes

Credit:
This issue was reported to the Apache Software Foundation via JPCERT.

References:
[1] http://markmail.org/message/kpfl7ax4el2owb3o
[2] http://tomcat.apache.org/security-8.html
[3] http://tomcat.apache.org/security-7.html
```

Fig. 1: Example of a security advisory related to `CVE-2014-0050` that was posted in the Apache common developers mailing list.

(ii) *Semantic versioning of releases* - The semantic versioning naming convention[5] provides hints on the estimated migration effort needed to migrate a dependency. For instance, a major released version may require more migration effort than a minor released version of that library.

(2) A Security Advisory: A security advisory is an official public announcement of a verified vulnerable dependency. Security advisories are circulated through various mail forums, special mailing lists and security forums, with the key objective of raising developer awareness to these vulnerabilities. Figure 1 is an example of a mail announcement of the `CVE-2014-0050` vulnerability sent to Apache Open Source developers mailing list. Vendors and researchers keep track of each vulnerability through a tagged CVE Identifier (i.e., `CVE-xxx-xxxx`). Generally, the advisory contains the following information: (i) a description of the vulnerability, (ii) a list of affected dependencies and (iii) a set of mitigation steps, that usually includes a viable (patched) replacement dependency.

[5] `http://semver.org/`

We conjecture that for developers, the awareness of the security advisory is more important than the migration effort needed to migrate the vulnerable dependency. Therefore, we now introduce the role played by a security advisory in the life-cycle of a vulnerability. As defined by CVE, a vulnerability undergoes the following four phases:

(i) *Threat detection* - this is the phase where the vulnerability threat is first discovered by security analysts.
(ii) *CVE assessment* - this is the phase where the threat is assessed and assigned a rating by the CVE.
(iii) *Security advisory* - this is the phase where the threat is publicly disclosed to awareness mechanisms such as the US National Vulnerability Database (NVD)[6] to gain the attention of maintainers and developers.
(iv) *Patch release* - this is the phase where authors of the vulnerable library provide mitigation options, such as a viable dependency that mitigates the threat. Once a viable replacement dependency (i.e., patch release) becomes available, developers can then proceed to complete the library migration process.

There are some cases were developers may have migrated their vulnerable dependency before the security advisory. We find that in these cases the vulnerability life-cycle does not synchronize with the migration process, causing the patch release to become available before the security advisory has been disclosed to the general public.

3 Research Questions

Our motivation stems from reports of outdated and vulnerable libraries being widespread in the software industry. In 2014, Heartbleed[7], Poodle[8], Shellshock[9], –all high profile library vulnerabilities were found to have affected a significant portion of the software industry. In that same year, Sonatype determined that over 6% of the download requests from the Maven Central repository were for component versions that included known vulnerabilities. The company reported that in review of over 1,500 applications, each of them had an average of 24 severe or critical flaws inherited from their components[10].

Our main goals involve (1) investigation of whether or not library migration is practiced in real-world projects and (2) evaluation of developer awareness and the migration effort needed for a library migration. As a result, we designed three research questions that involves a rigorous empirical study and a follow-up survey on reasons why developers do not update their dependencies. Hence, we first formulate (RQ1) to investigate library migration in practice:

[6] `https://web.nvd.nist.gov/`
[7] `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160`
[8] `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-3566`
[9] `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6271`
[10] report published January 02, 2015 at `http://goo.gl/i8J1Zq`

Library Migration in Practice.

- **(RQ1)** *To what extent are developers updating their library dependencies?* Prior studies have shown that a developer response to a library update opportunity is slow and lagging. Overall, related studies analyzed library migration at the API level of abstraction. A study by Robbes et al. [2012] shows how projects from the Smalltalk ecosystem exhibited a slower reaction to API updates. Similar results were observed in studies performed for the Pharo [Hora et al., 2015] and Java [Sawant et al., 2016] programming language domains. Furthermore, Bavota et al. [2015] found the same results for projects that exist within the ecosystem of Apache products.
  We use (RQ1) to study library migration at a higher abstraction than the API level to better understand (i) the extent to which developers use third-party libraries and (ii) the migration trends of these libraries.

In addition to (RQ1), we are particularly interested in factors such as the effect of current awareness mechanisms and migration effort that affect library migration. Henceforth, we formulate (RQ2) and (RQ3) to investigate how developers respond to existing awareness mechanisms:

Developer Responsiveness to Awareness Mechanisms.

- **(RQ2)** *What is the response to important awareness mechanisms such as a new release announcement and a security advisory on library updates?* To fully utilize the benefits of a library, developers are strongly recommended to keep the latest and patched releases of their library dependencies. Therefore, in (RQ2) we study developer responsiveness to the awareness mechanisms of (i) new releases and (ii) security advisories.
- **(RQ3)** *Why are developers non responsive to a security advisory?* Studies have concluded that developers often *'struggle'* with change. They show that influencing factors such as personal opinions, organizational structure or technical constraints [Bogart et al., 2015, Plate and Ponta, 2015] determine whether or not a developer is likely to migrate a dependency. However, we conjecture that a vulnerable dependency warrants immediate action by developers. Therefore in (RQ3), we seek developer feedback to understand the reasons why developers do not migrate a vulnerable dependency. These reasons may include the awareness of the migration opportunity and the migration effort needed to facilitate the replacement dependency.

4 Research Methods

In this section, we present the research methods used to address each of the three research questions. To answer (RQ1), we conducted an empirical study by mining and reconstructing historic library migrations for a set of real-world projects. To answer (RQ2), we then study 8 library migration cases to understand how developers respond to the awareness mechanisms of new releases and security advisories. Finally, to answer (RQ3), we interviewed developers belonging to projects with vulnerable dependencies.

4.1 (RQ1) *To what extent are developers updating their library dependencies?*

Our research method to answer (RQ1) is through a vigorous statistical analysis of library migration for real-world projects. Our method is comprised of three steps: (1) tracking systems and dependency updates, (2) analysis of our proposed metrics and (3) data collection. The results of (RQ1) are presented in Section 5.

(1) Tracking System and Library Updates: To accurately track library migrations, we define a model of system and library dependency relations. Hence, we formally use the following notations. We distinguish and define $\mathcal{S}$ for a system, and $\mathcal{L}$ for a library. The notation $\mathcal{L}(\texttt{lib},v)$ denotes version v of the library `lib`, while the notation $\mathcal{S}(\texttt{sys},w)$ refers to version w of the system `sys`. When a system version $\mathcal{S}(\texttt{sys},w)$ migrates to library version $\mathcal{L}(\texttt{lib},v)$, it creates a dependency relation between them.

Figure 2 illustrates how we use the notation to represent dependency relations between systems and libraries over time. The example in this figure consists of the following system and library versions:

- Library `A` which has 1 version $\mathcal{L}(\texttt{A},\mathit{1})$.
- System `B` which has 2 versions $\mathcal{S}(\texttt{B},\mathit{1})$ and $\mathcal{S}(\texttt{B},\mathit{2})$.
- Library `C` which has 2 versions $\mathcal{L}(\texttt{C},\mathit{1})$ and $\mathcal{L}(\texttt{C},\mathit{2})$.
- System `D` which has 3 versions $\mathcal{S}(\texttt{D},\mathit{1})$, $\mathcal{S}(\texttt{D},\mathit{2})$ and $\mathcal{S}(\texttt{D},\mathit{3})$.

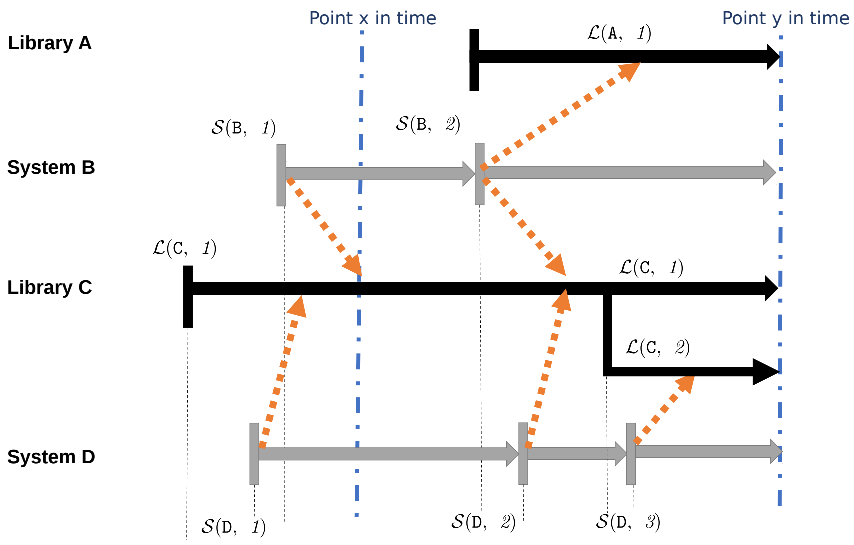


Fig. 2: Library migration between systems and libraries. The orange arrow depicts dependency relations between them.

Figure 2 then depicts the following library dependency relationships as an orange dotted line. We list below all dependency relations changes that are because of a library migration:

– System B (i.e., $\mathcal{S}$(B,*2*)) migrates to the library dependency $\mathcal{L}$(A,*1*).
– System B (i.e., $\mathcal{S}$(B,*1*) and System D (i.e., $\mathcal{S}$(D,*1*)) both migrate to the library dependency $\mathcal{L}$(C,*1*).
– System D(i.e., $\mathcal{S}$(D,*3*)) migrates to the library dependency $\mathcal{L}$(C,*2*).

From a system dimension, our model can track how often developers migrate their libraries. Since the release of a system version may contain multiple migrations, the model allows us to track the number of migrations that have occurred during one system version update, which is denoted as DU.

Dependency Update (**DU**) is the count of library migrations in a system version update.

Figure 2 depicts an example of a DU update (i.e., DU=1). We can see in the figure that for the $\mathcal{S}$(B,*2*) release, the system migrates to a new dependency (i.e., $\mathcal{L}$(A,*1*)) while still keeping the same version of its other dependency (i.e., $\mathcal{L}$(C,*1*)).

From the alternative library dimension, our model is able to track library usage trends over time. Our model can track the number of migrations that have occurred within the universe of known systems to determine the usage of a library, which is denoted as LU.

Library Usage (**LU**) is the total population count of user systems having a dependency relation to a library dependency at a specific point in time.

Figure 2 shows an example of the LU metrics. The figure shows that at point x in time, the LU of the $\mathcal{L}$(C,*1*) dependency is two (i.e., systems B and D). However at point y in time, since the $\mathcal{S}$(D,*3*) system release has migrated to the $\mathcal{L}$(C,*2*) dependency, updating the LU counts of the $\mathcal{L}$(C,*1*) and $\mathcal{L}$(C,*2*) library versions to one dependent system each. Furthermore, a system can use older versions of a library. This is modeled and shown in the figure, as a line branching out from the original line of library versions. For instance, library C separates into two different branches because the older $\mathcal{L}$(C,*1*) dependency still has an active dependency relation to one system (i.e., system B).

(2) Analysis Method: Table 1 provides a summary of the metrics provided by our model. To fully understand the migration phenomena and address (RQ1), we use our model to propose metrics that can analyze library migrations from both the system and library dimensions.

From the system dimension, we analyze our system metrics to investigate the distribution of dependencies per system (m1) and the frequency of library migrations per library (m2). First, we utilize boxplots and descriptive statistics to report the median ($\bar{x}$) and mean (μ) for each metric. We then test the hypothesis that *'systems with more dependencies tend to have more frequent*

Table 1: Summary of our proposed migration metrics defined for (RQ1). Note: we use Dep. = Dependencies

Alias	Dimension	Metric	Brief Description
m1	System	Dep. Per System (#Dep.)	# of Dependencies
m2		Dep. Update Per System (DU)	# of Dependencies migrated
m3	Library	Library Usage(LU)	# of systems
m4		Peak LU	max. # of systems
m5		Current LU	current # of systems
m6		Pre-Peak	time to reach Peak LU
m7		Post-Peak	time after Peak LU
m8		Library Residue	% remaining systems after Peak LU

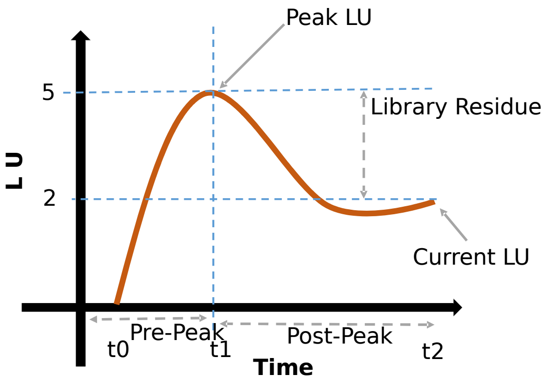


Fig. 3: Simple example of the LU-based metrics. We show the Peak LU at time t1, current LU at time t2 and Library Residue (Peak LU / Current LU).

updates'. This is done by employing the Spearman and Pearson correlation tests [Edgell and Noon, 1984] to test and determine any correlation relation between metrics `m1` and `m2`. A high correlation score confirms the assumption that a more complex systems will tend to have more updates, while a low correlation will confirm the hypothesis that *'the number of library dependencies does not influence the frequency of updates'*.

From the library dimension, we investigate how the migration away from a specific library dependency spreads over time, which is defined as the LU trend. This work is inspired by the Diffusion of Innovation curves [Rogers, 2003], which seeks to explain how, why, and at what rate new ideas and technology spreads. Figure 3 is a visual example of the LU metrics from Table 1. We utilize the LU metrics to study the LU trends of (i) whether or not a library dependency is gaining or losing system users) and (ii) the rate of decline after system users begin to migrate away from that dependency. We use Figure 3 to introduce a simple example of how we characterize a LU trend:

– *LU counts* - The *Peak LU* (`m4`) metric describes the maximum population count of user systems reached by a dependency. The *Current LU* (`m5`) is

a related metric that describes the latest population count of user systems that actively use this dependency in their systems.
- *LU over time* - The *Pre-Peak* (m6) metric refers to the time taken for a dependency to reach a peak LU (days). Conversely, *Post-Peak* (m7) metric refers to the time passed since the peak LU was reached (i.e., measured in days).
- *LU rate after Peak LU* - The *Library Residue* (m8) metric describes the percentage of user systems (%) remaining after *Peak LU* (m4) has been reached for a dependency (i.e., Current LU (m5) / Peak LU (m4)).

In Figure 3, we show the LU metrics as a LMP curve. In detail, we find that the *Peak LU* is 5 users at t1, with the *current LU* at 2 users. At the starting point $t0$, *Pre-Peak* is the period from $t0$ to $t1$ and *Post-Peak* being the time from $t1$ to $t2$. Quantitatively, we conjecture that a low Library Residue (i.e., 40% (2/5)) indicates that any affected developers using this dependency should consider migration towards a replacement dependency.

We present four statistical analysis to report the LU trends of our study. First, we use a cumulative frequency distribution graph to understand the distribution of popular library versions (m4 and m5). We then use a cumulative distribution to measure the average time for libraries to reach their peak usages (m6 and m7). Third, we use boxplots to measure the distribution of the Library Residue metric (m8). The final analysis is to plot and analyze the number of system dependencies along with their Library Residue.

(3) Data Collection: It is important that we test our approach from a quality set of real-world projects to improve the confidence on our results. Therefore, we conducted a large-scale empirical evaluation of software systems and library migrations that focuses on popular Java projects that use Maven libraries as their third-party dependencies. In detail, we collected projects that reside in GitHub[11] as the source of our dataset. To ensure that our dataset is a quality representation of real-world applications, we enforced the following pre-processing data quality filters:

- *Projects that are mature and well-maintained* - The first quality filter is to ensure that migrations are indicative of active and large-scale projects that are hosted on GitHub (i.e., removing toy projects). Hence, we selected projects that had more than 100 commits and had at least a recent commit between January 2015 and November 2015.
- *Projects that are unique and not duplicates* - The second quality filter is to ensure that no duplicates exist within the collected dataset. Hence, we semi-automatically inspected repository names to validate that none of the projects are forks from other projects (i.e., same project name in different repository).
- *Projects that use a dependency management tool* - We conjecture that projects managed by a dependency management tool is more likely to perform library migrations. Therefore, the third filter distinguishes projects

[11] https://github.com/

that implement a dependency management tool such as the Maven dependency management tool. Every Maven project includes a Project Object Model file (i.e., `pom.xml`) that describes the project's configuration metadata —including its compile and run time library dependencies.

Listing 1: Code snippet of the `pom.xml` metafile for the `GitWalker` system showing the dependency relationship to between two Maven dependencies, `javaparser` and `gitective-core`.

```
...
<groupId>GitWalker</groupId>
<artifactId>GitWalker</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>GitWalker</name>
...
<dependencies>
  <dependency>
    <groupId>com.google.code.javaparser</groupId>
    <artifactId>javaparser</artifactId>
    <version>1.0.8</version>
  </dependency>
  <dependency>
    <groupId>org.gitective</groupId>
    <artifactId>gitective-core</artifactId>
    <version>0.9.9</version>
  </dependency>
</dependencies>
...
```

Listing 1 shows a `pom.xml`, which lists dependency relationships between a particular system version with any valid Maven library version. In this example, we extract the dependency relation for system $\mathcal{S}(\texttt{Gitwalker}, \textit{0.0.1-SNAPSHOT})$ that uses the $\mathcal{L}(\texttt{javaparser}, \textit{1.0.8})$ and $\mathcal{L}(\texttt{gitective-core}, \textit{0.9.9})$ dependencies. To automatically extract the history of dependency migrations for a project, we analyze historic changes of the `pom.xml`. We package this extraction method in a tool called PomWalker[12].

– *Popular and latest dependency versions* - LU trends require sufficient usage by systems. As a result, we focus on the more popular libraries for a higher quality result. Moreover, to capture migrations away from a library dependency, we filter out the latest versions of any library in the dataset.

Table 2 presents a summary of the filtered 4,659 projects after preprocessing from an original collection of 10,523 GitHub projects. Our study tracks dependency migration between a Maven library and each unique system within each project (i.e., a project may contain multiple systems). We then mine 48,495 systems from the 4,659 software projects to extract 852,322 dependency migrations. For the LU trend analysis, we filter out rarely used libraries (i.e., dependencies with less than 4 user systems are defined as unpopular) and the latest library versions (i.e., we assume that the 213 latest

[12] `https://github.com/raux/PomWalker`

Table 2: Summary of the collected dataset

	Dataset statistics
projects creation dates	2004-Oct to 2009-Jan
projects last update	2015-Jan to 2015-Nov
# unique systems (projects)	48,495 (4,659)
# unique library versions	2,736
total size of projects	630 GB
# commits related to `pom.xml`	4,892,770
# library dependency migrations	852,322

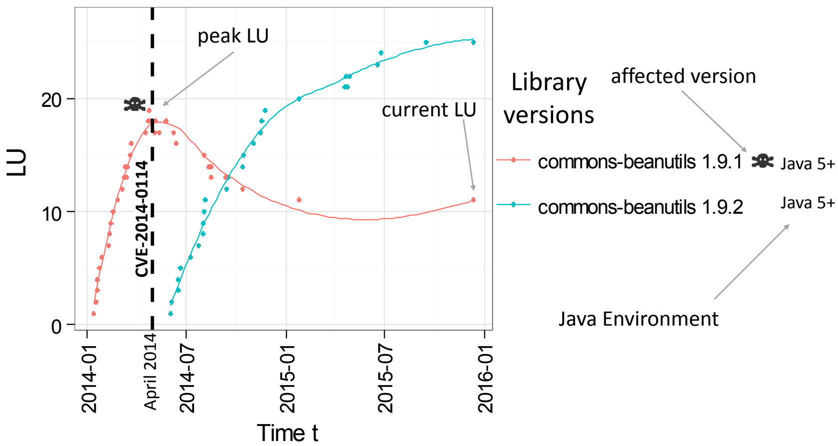


Fig. 4: A Library Migration Plot for libraries $\mathcal{L}$(`beanutils`,*1.9.1*) and $\mathcal{L}$(`beanutils`,*1.9.2*). In this example, the release of a related security advisory `CVE-2014-0114` (black dashed line) that affects $\mathcal{L}$(`beanutils`,*1.9.1*) (marked with crossbones). We also show which JDK (5+) version in which the version supports.

versions are less likely to experience any migrations), which leaves 2,736 library versions available for our study.

4.2 (RQ2) *What is the response to important awareness mechanisms such as a new release announcement and a security advisory on library updates?*

Our method to answer the second research question (RQ2) is through a case study analysis of developer responsiveness to existing awareness mechanisms. It is comprised of three steps: (1) tracking library migration in response to awareness mechanisms (2) analysis method and (3) data collection. The selected case studies for the new release announcement are presented in Section 6.1, with cases for the security advisory presented in Section 6.2.

(1) Tracking Migration in Response to Awareness Mechanisms: Figure 4 presents the Library Migration Plot (LMP) used to track LU trends over time. Together with the change log and semantic versioning documentation, we use LMPs to infer library migration patterns and trends. The LMP shows LU changes in the library (y-axis) with respect to time (x-axis). It is important to note that the LMP curve itself should not be taken at face value, as the smoothing algorithm is generated by a predictive model and it is not a true reflection of all data points. We observe in Figure 4 that the `commons-beanutils` library $\mathcal{L}$(`commons-beanutils`,*1.9.1*) (red line) had 19 user systems using it as a dependency in April 2014. By January 2015, its LU had decreased to 11 user systems. In particular, we annotate the following events onto the LMP to depict the effect of awareness mechanisms on the LMP curve:

– *Official Release Announcement* - Figure 4 depicts an example of two versions: $\mathcal{L}$(`commons-beanutils`,*1.9.1*) and $\mathcal{L}$(`commons-beanutils`,*1.9.2*). Hence, we can use the LMP to compare the migration patterns between versions of a library. For instance, the LMP presents the effect of the new release of $\mathcal{L}$(`commons-beanutils`,*1.9.2*), illustrated by the declining LU curve at $\mathcal{L}$(`commons-beanutils`,*1.9.1*).
– *Security Advisory Disclosure* - Figure 4 annotates when the security advisory `CVE-2014-0114` was disclosed to the public (i.e., April 2014). In detail, the LMP presents evidence of how a security vulnerability triggers the library migration from $\mathcal{L}$(`commons-beanutils`,*1.9.1*), illustrated by its declining LU curve.

(2) Analysis Method: Our approach to answer (RQ2) involves a manual case study analysis to understand developer responsiveness to a new release announcement and a security advisory. For more useful and practical scenarios, selection of our case studies included (i) new releases from the more popular libraries (i.e., as they tend to impact more developers) and (ii) more severe security advisories (i.e., warrants immediate developer attention).

At the quantitative level, we first visually analyze the LMP curve response towards the awareness mechanism. We then manually consult online documentation such as the release logs, and its semantic versioning schema to estimate the effort needed to migrate towards a newer replacement dependency. For the vulnerable dependencies, we consult information from the security advisory and the life-cycle of a vulnerability (See Section 2.2) to estimate the needed migration effort. For example, in Figure 4, we infer from the release notes that $\mathcal{L}$(`commons-beanutils`,*1.9.1*) to $\mathcal{L}$(`commons-beanutils`,*1.9.2*) update is a compatible minor update with 2 bug fixes and 1 new feature. Since both are supported by the latest JDK (Java 5 and higher), we assume that the migration effort required is much lower when compared to a dependency that supports a different JDK environment.

(3) Data Collection: Since our research method to answer (RQ2) is through the use of case studies, we systematically select a subset of eligible projects from the dataset collected in (RQ1). Selection of a new release candidate is comprised of three steps. First, since our objective is to find common LU

Table 3: Top 20 LU library versions

	Library	Versions
*	junit	(4.11), (4.10), (4.8.2), (3.8.1), (4.8.1)
	javax.servlet-servlet-api	(2.5)
	commons-io-commons-io	(2.4), (2.6)
*	log4j-log4j	(1.2.16), (1.2.17)
	commons-lang	(2.6)
	commons-logging	(1.1.1)
	commons-lang	(3-3.1)
	commons-collections	(3.2.1)
	javax.servlet-jstl	(1.2)
	org.mockito-mockito-all	(1.9.5)
	commons-httpclient	(3.1)
*	guava	(14.0.1), (18.0)
	commons-dbcp	(1.4)

Table 4: New Release case studies from three popular libraries. For each library, we look at the LU trends of three libraries.

Alias	Library	ver.1	ver.2	ver.3
NR1	`google-guava`	16.0.1 (2014-02-03)	17.0 (2014-04-22)	18.0 (2014-08-25)
NR2	`junit`	3.8.1 (2002-08-24)	4.10 (2011-09-29)	4.11 (2012-11-15)
NR3	`log4j`	1.2.15 (2007-08-24)	1.2.16 (2010-04-06)	1.2.17 (2012-05-06)

Table 5: Security Advisory case studies from the Apache Software Foundation of Maven libraries. Note that the affected versions include all prior versions. Likewise safe versions also include all superseding versions.

Alias	CVE Id	library	Release	Affected ver.	Vulnerability(CVSS)
V1	CVE-2014-0114	`commons-beautils`	2014-04-30	1.9.1	Denial of Service (7.5)
V2	CVE-2014-0050	`commons-fileupload`	2014-01-04	1.3	man–in–the–middle(5.8)
V3	CVE-2012-5783	`commons-httpclient`	2012-04-11	3.x	man–in–the–middle(4.3)
V4	CVE-2012-6153	`httpcomponents`	2014-09-04	4.2.2	man–in–the–middle(7.5)
V5	CVE-2012-2098	`commons-compress`	2012-06-29	1.4	man–in–the–middle(5.0)

trends popular libraries, we select the top 20 library versions out of the 2,736 available from our dataset. The top 20 libraries are shown in Table 3. For each of the 20 library versions, we then generate and categorize them based on LMP curve patterns. In the final step, we select three case studies that depict distinctive LU trends. Table 4 shows the nine popular library versions of `google-guava`[13], `junit`[14] and `log4j`[15] that met our selection criteria.

[13] `https://code.google.com/p/guava-libraries/`
[14] `http://junit.org/`
[15] `http://logging.apache.org/log4j/1.2/`

Table 5 shows the five security advisory cases that were analyzed in our study. The case study selection process involved choosing security advisories that (i) affect our collected dataset (i.e., in (RQ1)) and would (ii) require the immediate response of the developers. All cases were selected from a pool of CVE security advisories between 2009-2014. Particularly, we found 686 associated security advisories[16] that affected the 123 Apache Software Foundation (ASF) products. We then found that 15 out of the 123 ASF products are third-party libraries. Out of the remaining 15 libraries, we then chose five associated security advisories that are malicious enough to warrant a developer response. Specifically, the security advisory should have a medium to high Common Vulnerability Score (CVSS)[17] (i.e., 4 or higher). Table 5 also describes the malicious exposures of each security advisory: `V1` causes a *Denial of Service (DoS)* with a high CVSS score. The remaining four security advisory cases all describe web application exposure to a remote *'man in the middle'* web attack and has a medium-to-high CVSS severity rating.

4.3 (RQ3) *Why are developers non responsive to a security advisory?*

Our research method to answer (RQ3) is through a survey of affected developers that were non responsive to a severe security advisory (i.e., identified in (RQ2)). Our method makes use of a qualitative survey interview form and comprises of two steps: (1) survey design and (2) data collection. Results to (RQ3) are presented in Section 6.3.

Listing 2: Email snippet of the survey form sent to developers of the selected projects that were non responsive to a security vulnerability.

```
<!---email snippet/>
Dear GitHub OSS Developer,
...
As a part of my study I particularity focused on the <library version/> and
the <CVE-xxx-xxxx/> <CVE URL/>, announced on <date>, which affects versions xxx.
We noticed that your project on GitHub is still configured to depend on a
vulnerable version of <library version/> at <https://xxx.xxx.xx/pom.xml/>
We understand that there are many reasons for not migrating, thus we appreciate
if you could simply detail the following:
1. Were you aware of the vulnerability? If so, then how long ago.
2. What are some factors that influence you not to update?
...
<!---email snippet/>
```

(1) Survey Design: Listing 2 shows the template of our survey form[18] sent to developers of the contactable projects. We targeted projects that allowed

[16] An updated listing is available online at `http://www.cvedetails.com/product-list/vendor_id-45/apache.html`

[17] it is officially known as the CVSS v2 base score. The calculation is shown at `https://www.first.org/cvss/v2/guide`

[18] the complete form is available at `http://sel.ist.osaka-u.ac.jp/people/raula-k/librarymigrations/questionaire.html`

public communication, either through an issue management system or a mailing list. The survey form is designed with two parts. First, we customized the survey form to include project specifics, such as the exact location of the `pom.xml` file where the dependency is being relied upon. The second part of the survey then asked developer opinions on the following two questions: (i) *Were you aware of the vulnerability? If so, then how long ago* and (ii) *What are some factors that influence you not to update?*

For the analysis, we first tally the survey responses according to whether or not the developer was aware of vulnerable dependency in their systems. We then analyze the responses through a systematic method of (i) reading of each response, (ii) checking and summarizing text by consistency, omissions and (iii) looking for similarities or differences between interviewee responses. In detail, we perform analysis of the responses in three steps. First, the main author categorizes all responses based on the systematic method. Another author is then tasked to verify and criticize each category of responses. In the final step, the categories are then presented to rest of the authors for a group consensus.

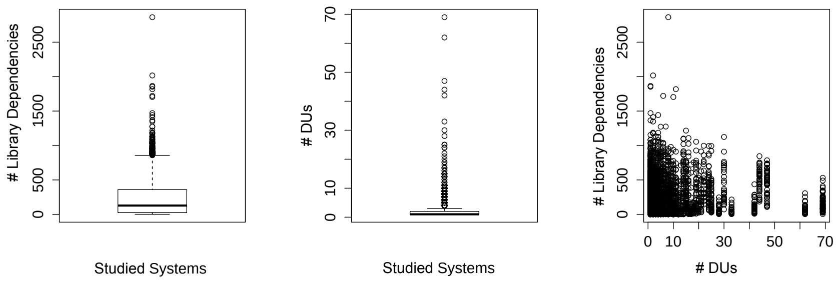
***(2) Data Collection**:* For (RQ3), we use the five case studies in (RQ2) to identify and survey all projects that did not respond to each of the security advisory. From the LMP analysis in (RQ2), we are able to identify the candidate projects that were found to be non responsive to the security advisory announcement. Out of all candidate projects, we collected 16 developer responses. Due this amount of responses, the analysis method was managed by one author and then later criticized and verified by other authors for the final consensus. All results of the collected dataset, including the tally of listed and contactable projects are presented in Section 6.3.

5 Library Migration in Practice

In this section, we present the results for (RQ1) *To what extent are developers updating their library dependencies?* In detail, we present the statistical results from both a system (Section 5.1) and a library dimension (Section 5.2), before finally answering (RQ1).

5.1 System Dimension

Figure 5 shows the results on how maintainers manage and update their dependencies from a system viewpoint. Specifically, the distribution of library dependencies per system in Figure 5(a) confirms that systems show heavy dependence on libraries ($\bar{x}$=147, μ=267.2, σ=311.56). A reason for this heavy reliance on libraries is because many of the analyzed projects are comprised of multiple subsystems that together form a complex set of dependencies. Furthermore, Figure 5(b) suggests that systems rarely update library dependencies, statistically we find low frequency of DU per system (i.e., $\bar{x}$=1, μ=2.4),



(a) # dependencies per system (b) # DU per system (c) # dependencies vs. # DU

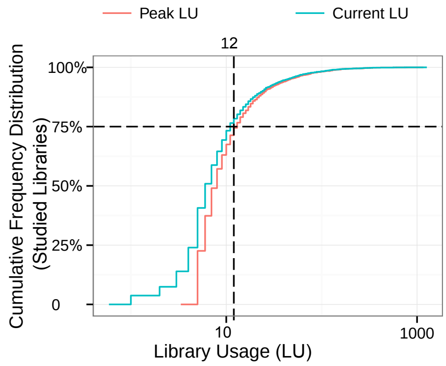
Fig. 5: Updates from a System dimension depicts (a) # of dependencies per system. ($\bar{x}$=147, μ=267.2, σ=311.56) (b) frequency of DUs per system ($\bar{x}$=1, μ=2.4, σ=4.2) and (c) relationship between # of dependencies vs. # of DUs (log-scale).

and each DU containing at least two library dependencies (i.e., $\bar{x}$=2, μ=4.1, σ=14.9). According to Figure 5(c), the findings do not show a strong correlation between the number of library dependencies and the frequency of DU, with statistical tests reporting weak correlations (pearson = 0.05, spearman = 0.07). This result confirms the hypothesis that the *number of library dependencies in a system does not influence the frequency of updates.*

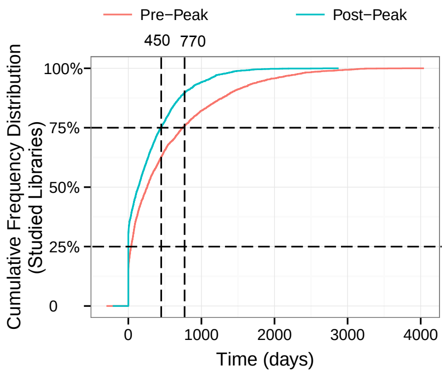
5.2 Library Perspective

Figure 6 and Figure 7 both present the LU trend analysis of our studied systems. Figure 6(a) shows that LU for 75% of the popular libraries is 12 (i.e., peak LU). Interestingly, we found that 596 libraries exhibited no library migration movement, which is indicated by the peak library usage being also the current library usage (i.e., peak LU = current LU). Figure 6(b) also shows that reaching the peak library usage is slow for most dependencies. Furthermore, the figure shows that 25% of dependencies took less than a day to reach their peak LU. Afterwards the rate slows down (depicted by curve), showing 75% of dependencies took less than 770 days to reach their peak LU (i.e., Pre-Peak). Upon closer inspection, we found that these dependencies were specialized libraries that were used by a smaller number of systems (i.e., low LU).

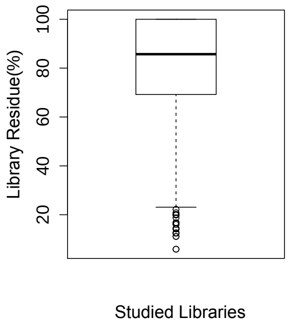
After reaching peak usage, most dependent systems tend to slowly migrate away. As shown in Figure, we find that 75% of library dependencies experience some migration of its users over the next 450 days (ie., Post-Peak). Importantly, Figure 6(c) suggests that many systems remain with an outdated dependency, even after some library migration away from the dependency has begun. The figure also shows that most of the 2,736 studied dependencies exhibit high Library Residue (i.e., $\bar{x}$=85.7%, μ=81.5%, σ=22.2%). An example



(a) LU Distributions per dependency



(b) Time-frame analysis



(c) Library Residue per dependency

Fig. 6: Updates from a Library dimension depicts the cumulative frequency distribution (a) of Peak LU and Current LU (Log scale), (b) time-frame metric distributions and the boxplot of (c) Library Residue (%) for 2,736 dependencies.

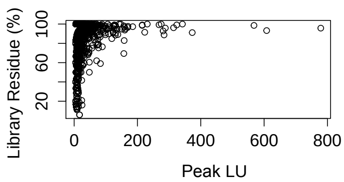


Fig. 7: A correlation of Library Residue against Peak LU, showing that popular library dependencies (with higher peaks) also tend to exhibit higher Library Residue.

is the popular but older `log4j` logging library $\mathcal{L}$(`log4j`,*1.2.15*) which has a Library Residue of 98%. Finally, Figure 7 shows that the system are more likely to remain with the more popular libraries, with higher peaking libraries exhibiting more Library Residue. Based on our results, we now return to answer (RQ1):

> We conducted an empirical study to understand the extent to which (i) systems use and manage their library dependencies and (ii) library usage trends. To answer (RQ1): *(i) although system heavily depend on libraries, most systems rarely update their libraries and (ii) systems are less likely migrate their library dependencies, with 81.5% of systems remaining with a popular older version.*

6 Developer Responsiveness to Awareness Mechanisms

We present results in Section 6.1 and Section 6.2 for (RQ2) *What is the response to important awareness mechanisms such as a new release announcement and a security advisory on library updates?* while our results for (RQ3) *Why are developers non responsive to a security advisory?* are presented in Section 6.3. Table 6 shows the aliases (i.e., NR1, ..., NR3, V1, ..., V5) used as a reference to each of the case studies.

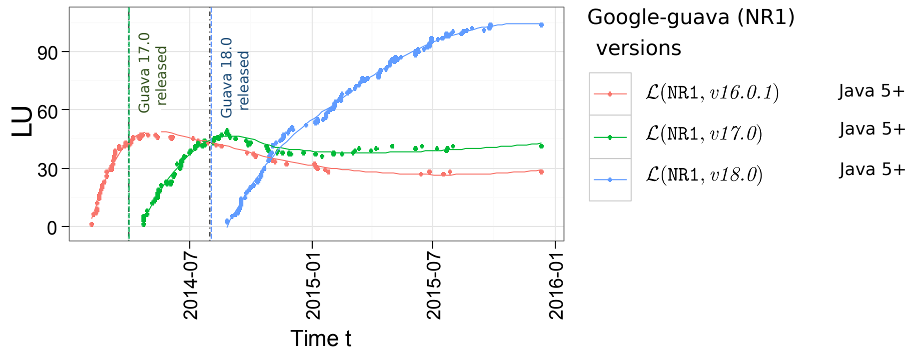
6.1 A New Release Announcement

Figure 8 depicts our case studies (NR1, NR2, NR3) related to responsiveness of a new release, with (A) consistent and (B) non responsive library migration trends.

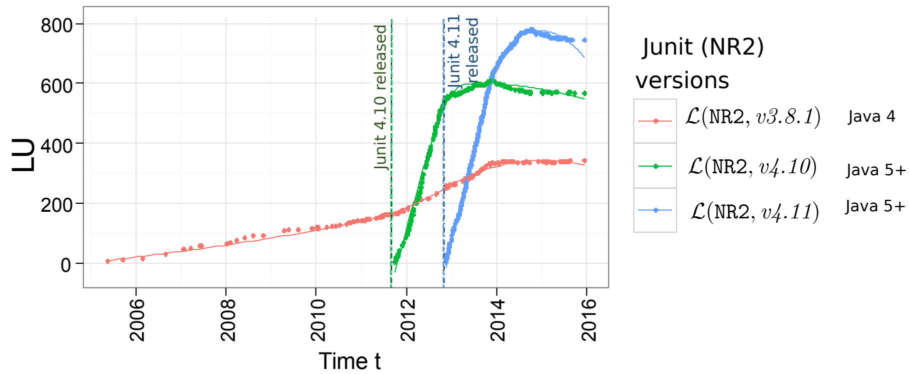
(A) Cases of an Active Developer Response to a New Release. Figure 8(a) shows an example of library that have a consistent library migration trend. Specifically, the LMP of `google-guava` (NR1) $\mathcal{L}$(`NR1`,*16.0.1*) and $\mathcal{L}$(`NR1`,*17.0*) depict a consistent pattern of migration with 48 and 49 peak LU. This pattern is consistent, despite the libraries having a relatively high Library Residue of 60.4% and 85% for all studied versions.

Table 6: Alias names for our (RQ2) selected case studies.

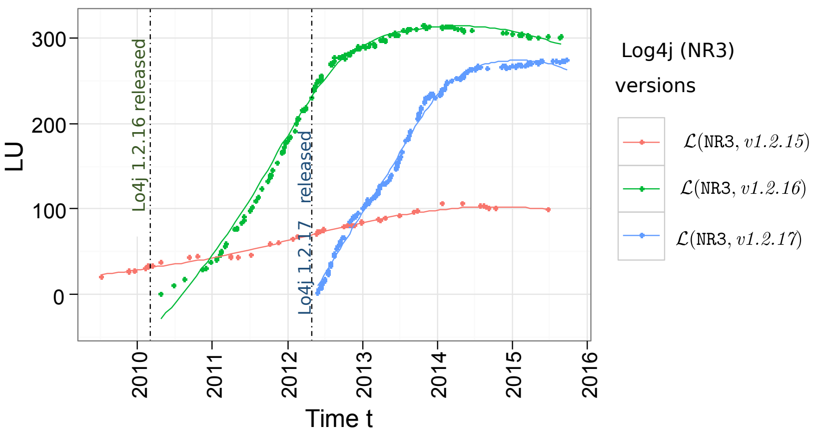
Alias	Awareness Mechanism	Library	Analyzed versions
NR1	New Release	google-guava	(16.0.1), (17.0), (18.0)
NR2		junit	(3.8.1), (4.10), (4.11)
NR3		log4j	(1.2.15), (1.2.16), (1.2.17)
V1	Security Advisory	commons-beautils	(1.9.1), (1.9.2)
V2		commons-fileupload	(1.2.2), (1.3), (1.3.1)
V3		commons-httpclient	(3.1), (4.2.2)
V4		httpcomponents	(4.2.2), (4.2.3), (4.2.5)
V5		commons-compress	(1.4), (1.4.1)



(a) LMP for consecutive releases of the `google-guava` (NR1) library



(b) LMP for consecutive releases of the `junit` (NR2) library



(c) LMP for consecutive releases of the `log4j` (NR3) library

Fig. 8: Library Migration Plots (LMP) of three libraries depicting successive library version releases without vulnerability alerts.

We find that the reasons for consistent migration trends are mainly related to the estimated migration effort required to complete the migration process. Through inspection of the online documentation, we find that migration from $\mathcal{L}$(NR1,*16.0.1*) to $\mathcal{L}$(NR1,*17.0*) contains 10 changed packages[19]. Similarly, migration from $\mathcal{L}$(NR1,*17.0*) to $\mathcal{L}$(NR1,*18.0*) also contained 7 changed packages. Yet, all three library versions require the same Java 5 environment which indicates no significant changes to the overall architectural design of the library. From the documentation, we deduce that popular use of $\mathcal{L}$(NR1,*18.0*) is due to the prolonged period between the next release of $\mathcal{L}$(NR1,*19.0*), which is more that a year after the release of $\mathcal{L}$(NR1,*18.0*) in December 10, 2015. In fact, previous versions had shorter release times, around 2-3 months of $\mathcal{L}$(NR1,*16.0.1*) in February 03 2014, $\mathcal{L}$(NR1,*17.0*) in April 22 2014, and $\mathcal{L}$(NR1,*18.0*) in August 25 2014. The prolonged released cycles of the library could be related to the relatively higher peak LU of $\mathcal{L}$(NR1,*18.0*) at 100 LU compared to the lower peaks LU of $\mathcal{L}$(NR1,*16.0.1*) at 48 LU and 49 LU for the $\mathcal{L}$(NR1,*17.0*) dependency.

(B) Cases of a Developer Non Response to a New Release. Figure 8(b) depicts *'lack of developer response'* reaction to a dependency migration opportunity. The LMP curve from figure depicts the older popular versions as exhibiting no migration movement (i.e., peak LU= current LU). Specifically for the `junit` (NR2) library, the dependency $\mathcal{L}$(NR2,*3.8.1*) does not follow the typical migration pattern of the $\mathcal{L}$(NR2,*4.10*) and $\mathcal{L}$(NR2,*4.11*) dependencies.

Similar to the consistent migration to a new release, we find that the reason for a non response to a migration opportunity is related to the estimated migration effort. For instance, as shown in Figure 8(b), the newer `Junit` version 4 series libraries requires a change of platform to Java 5 or higher ($\mathcal{L}$(NR2,*4.10*) and $\mathcal{L}$(NR2,*4.11*)), inferring significant changes to the architectural design of the library. Intuitively, we see that even though $\mathcal{L}$(NR2,*3.8.1*) is older, it still maintains its maximum library usage (i.e., current LU and peak LU=342). This LMP curve pattern is also apparent in the `log4j` (NR3) library shown in Figure 8(c), with the $\mathcal{L}$(NR3,*1.2.15*) dependency being older, but still active library version (i.e., with over 100 current LU). We visually observe that as $\mathcal{L}$(NR3,*1.2.17*) dependency reaches its peak LU the $\mathcal{L}$(NR3,*1.2.16*) dependency remains more popular, with a higher LU than superseding library release. This result complements the findings in (RQ1) that popular library dependencies tend to retain most of their users, even if a possible migration to a new release opportunity is available.

6.2 Security Advisory Disclosure

Figures 9, 10, 11 and 12 all depict the LMP of our case studies related to the responsiveness of affected developers towards a security advisory disclosed

[19] details at `http://google.github.io/guava/releases/17.0/api/diffs/`

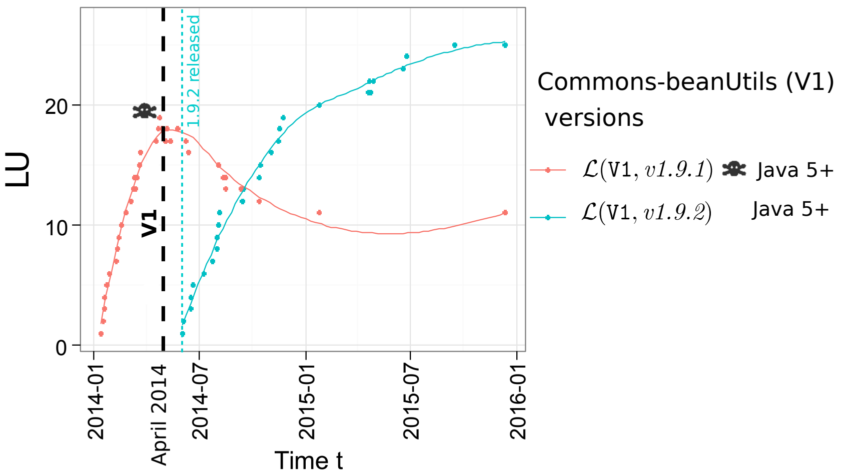


Fig. 9: LMP for vulnerability V1, related to the COMMONS-BEANUTILS library dependency versions $\texttt{V1}_{1.9.1}$ and $\texttt{V1}_{1.9.2}$.

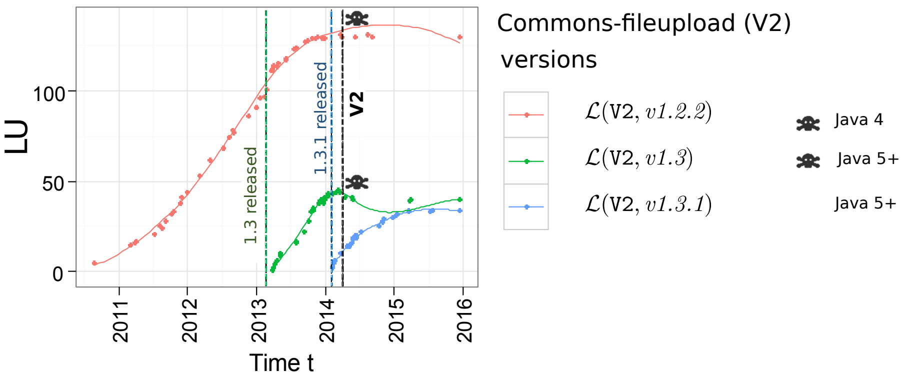


Fig. 10: LMP for vulnerability V2, related to the `commons-fileupload` library dependency versions.

to the general public. In our analysis, we group and discuss the case studies according to (C) an active response, (D) a lack of response and (E) a latent disclosure of the security advisory.

(C) Cases of an Active Developer Response to a Security Advisory Disclosure. Figure 9 depicts a typical case of where migration is in response to a vulnerability. As shown, the LMP curve clearly depicts a peak and decline in the usage after the V1 vulnerability security advisory was disclosed to the public. We conjecture that the timely release of the patched $\mathcal{L}(\texttt{V1},\mathit{1.9.1})$ dependency shortly after the security advisory was disclosed, provided a migration opportunity for developers.

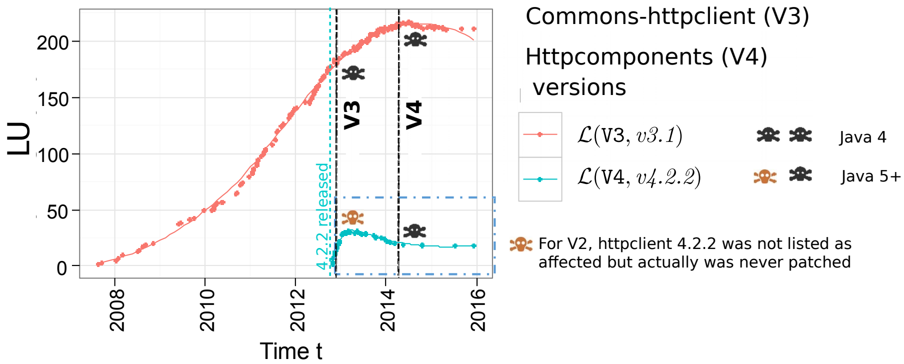
In contrast to the reported case in V1, Figure 10 depicts a case for V2, where the security for the V1 vulnerability security disclosure that affects the $\mathcal{L}$(V2,*1.3*) dependency does not affect the LMP curve of the older $\mathcal{L}$(V2,*1.2.2*) dependency. In detail, the LMP curve is evident by the rise of the $\mathcal{L}$(V2,*1.2.2*) dependency from 110 LU to 140 LU, during the period in which the $\mathcal{L}$(V2,*1.3.1*) dependency is released. Nevertheless, during this period $\mathcal{L}$(V2,*1.3*) having also increased from 1 to 48 LU during this period, inferring that during this period, maintainers preferred to adopt the older dependency rather than the newer release. The LMP curves in Figure 10 depict how the disclosure of the V2 security advisory does not trigger much migrating away from both the $\mathcal{L}$(V2,*1.2.2*) and $\mathcal{L}$(V2,*1.3*) vulnerable dependencies. We observe from the LMP that the Library Residue of these two libraries indeed high, with $\mathcal{L}$(V2,*1.2.2*) showing 98% Library Residue, while $\mathcal{L}$(V2,*1.3*) having 86% of Library Residue. The LMP infers that even though the security advisory has been disclosed to the public, many of these affected developers still continue to rely on these vulnerable dependencies in their projects.

As in the case of new release announcements, one reason why a developer may not respond to a security advisory is due to the estimated migration effort required. In the V2 vulnerability case, inspection of the release logs indicate a relatively high migration effort, as the newer $\mathcal{L}$(V2,*1.3*) dependency would require an upgrade towards the JDK platform of Java 5 or higher platform. Moreover, we conjecture that users of the newer $\mathcal{L}$(V2,*1.3*) dependency are more likely from developers that have not used prior versions of the affected commons-fileupload library.

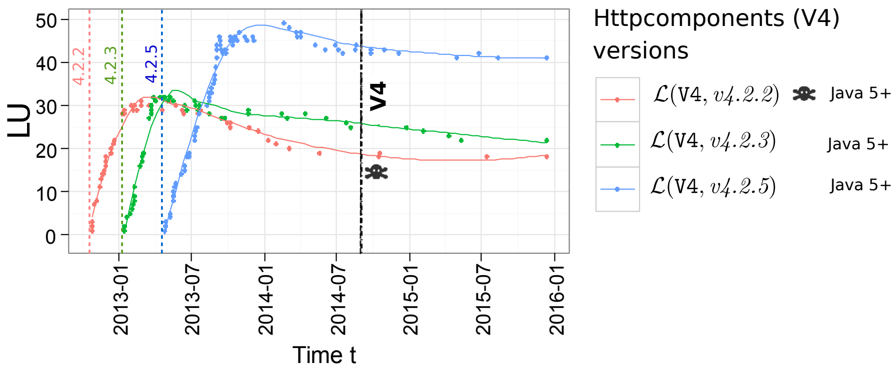
(D) Cases of an Incomplete Patch Release in Response to a Security Advisory Disclosure. Figure 11 shows a case where the lack of a replacement dependency may contribute to affected developers showing no response to the disclosure of a security advisory. In this case, the initial vulnerability V3 is related to the Amazon Flexible Payments Service (FPS), which is a man-in-the-middle attacker to spoof SSL servers via an arbitrary valid certificate. V3 is the original vulnerable $\mathcal{L}$(V3,*3.1*) dependency that affects users of the commons-httpclient library. As seen in Figure 11(a) rising LMP curve, the security advisory does not trigger any migration among its users. In fact, there is an increase from 165 to 209 LU after the security alert was disclosed. Related to V3, V4 is the same man-in-the-middle attach with a *'NOTE: this issue exists because of an incomplete fix for CVE-2012-5783'* in its description[20].

The estimated migration effort and the lack of a viable replacement dependency are some of the possible reasons why affected maintainers show no response to the security advisory. This is shown in the case of the Httpcomponents library[21], which is the successor and replacement for commons-httpclient library. As documented, Httpcomponents is a major upgrade with many architectural design modifications compared to the older

[20] https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-6153
[21] https://hc.apache.org/



(a) LMP for vulnerability V3, related to the `commons-httpclient` and the superseeding `httpcomponents` libraries for vulnerablilty V4. V4 is the fix for the V3 vulnerability that was not fixed for the vulnerable $\mathcal{L}$(V4,*4.2.2*) dependency.



(b) LMP for vulnerability V4, related to the `httpcomponents` library. Note: There seems no effect of V4, possibly because maintainers of the vulnerable $\mathcal{L}$(V4,*4.2.2*) dependency may have already migrated away to the safer $\mathcal{L}$(V4,*4.2.3*) and $\mathcal{L}$(V4,*4.2.5*) dependencies.

Fig. 11: Vulnerability alerts for the `commons-httpclient` (V3) and related `httpcomponents` (V4) library. In detail, Figure 11(b) is a zoomed in look at Figure 11(a), which is the vulenerable $\mathcal{L}$(V4,*4.2.2*) dependency, and safe $\mathcal{L}$(V4,*4.2.3*) and $\mathcal{L}$(V4,*4.2.5*) dependencies.

`commons-httpclient` dependency versions. However, after the first release of the `Httpcomponents` library, the LMP curve in Figure 11(a) indicates that many systems still actively use the older `commons-httpclient` (V3) version. Shown in this figure, after the *V4* security advisory disclosure, the affected $\mathcal{L}$(V4,*3.1*) showed signs of developers migrating away from the vulnerable dependency. The LMP curve of the $\mathcal{L}$(V4,*3.1*) dependency moves from a peak LU of 215 to a decreased 212 LU.

(E) Cases of a Latent Security Advisory Disclosure. Figure 11(b) shows a case where some affected developers had already migrated away from the vulnerable dependency, even before the security advisory was disclosed to the public. In the case of `httpcomponents` library, its LMP curve indicates that developers who maintain the vulnerable V4 $\mathcal{L}$(V4,*4.2.2*) dependency shows no

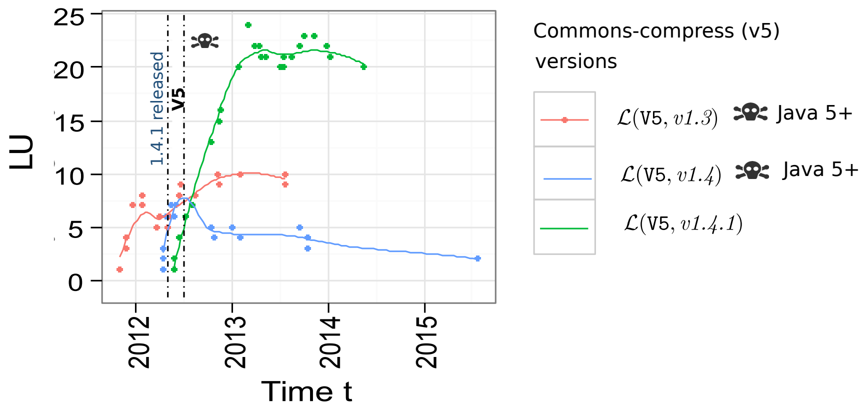


Fig. 12: LMP for vulnerability V5, related to the `commons-compress` library.

response to the security advisory disclosure. One reason is that a prior migration of the library had already been triggered by the releases of newer versions of $\mathcal{L}$(V4,*4.2.3*) and $\mathcal{L}$(V4,*4.2.5*). By the time V3 is disclosed, $\mathcal{L}$(V4,*4.2.2*) is already in decline with a 60% Library Residue.

Figure 12 depicts a case where the reason for developer responsiveness to a security advisory disclosure cannot be simply explained using the LMP curve. In the figure, the LMP curve shows that developers that maintain the vulnerable `commons-compress` $\mathcal{L}$(V5,*1.4*) dependency responded to the *V5* security advisory disclosure. However, this was not the case of all versions of the library. The LMP curve shows developers that use the older $\mathcal{L}$(V5,*1.3*) dependency did not show any signs of migrating away from this vulnerable dependency. In fact, although minor, the LMP curve for the vulnerable $\mathcal{L}$(V5,*1.4*) dependency rises in LU after the security advisory was disclosed to the public. Based on these results, we now return to answer (RQ2):

We conducted an empirical study to understand developer responsiveness to (i) a new release and (ii) a security advisory disclosure. To answer (RQ2): *we find that for a new release of a popular library (i) there exist patterns of consistent migration and patterns where an older popular library version is still preferred. For a security advisory disclosure we find cases of developer (ii) non responsiveness to security advisory disclosure, which is sometimes due to an incomplete patch or a latent security advisory. We find developers are less likely to update a library that requires more migration effort and vice-versa.*

6.3 Developer Feedback on Updating a Vulnerable Dependency

Table 7 shows a summary of affected projects that show negligence in responding to any of the five (i.e., V1, V2, V3, V4, V5) security advisories analyzed

Table 7: Summary of the survey collected from projects with a known security vulnerability.

Alias	# Listed	# Contactable	# Feedback	Unaware	Updated
V1	42	23	5	4	4
V2	40	26	6	6	5
V3	20	5	1	0	1
V4	10	7	3	1	0
V5	8	3	1	0	1
Totals	**120**	**64**	**16**	**11**	**11**

in (RQ2). From the 120 projects that were detected by the LMP curve, we found 64 of the projects provided a feedback mechanism such as a mailing list or issue management system. As shown, out of the 64 projects, 16 projects (25%) of the projects provided us feedback. In this section, we discuss the results pertaining to: (F) developer awareness on the vulnerability affecting their projects and (G) developer opinion regarding the practice of updating dependencies.

(F) Developer Awareness of Vulnerabilities in their Projects. Table 7 shows that many of the affected projects were unaware of the vulnerability to their software. Through the feedback, we find that out of the 16 responses, 11 (69%) immediately thanked us for the notification and proceeded to update their dependencies to the safer dependency versions.

(G) Developer Opinion on Library Updates. Developers cite the threat of the impact of the exposure as well as the function of the dependency as a factor to influence the decision in responding to a security advisory. A developer from a project responded that *'our project has been inactive and production has been halted for indefinite time'*. Developers from another two projects noted that the vulnerable dependency did not have a critical effect on the project:

> *'I knew about it because I happen to work on another project where we had to fix this very problem, but I didn't connect two dots. In this case, it's a test dependency, so the vulnerability doesn't really apply.'* and that *'It's only a test scoped dependency which means that it's not a transitive dependency for users of XXX so there is no harm done. XXX has no external compile scoped dependencies thus there is no real need to update dependencies.'*

Finally, the remaining two projects stated that the update was unnecessary as the affected component had little impact on the project objectives or part of their responsibilities. A developer from the first project stated that *'When it comes to this specific vulnerability org.apache.commons.httpclient is only used by XXX by the automatic update, and there's no SSL or encryption involved.'* while another developer from the latter project deferred the responsibility to another project *' We don't maintain XXX, but have passed this on to XXX. Not our decision, this is a slightly revised fork.'*

Similar to the results for (RQ2), all developers from the 16 projects cite the required migration effort as an influencing factor to whether or not a vulnerable library should be updated. For instance, one developer discusses how security updates are not a priority, as they do not align with the goals and objectives of their software customers:

> *'I subscribed to the CVE RSS recently and I don't check it regularly, so even if I might have heard of the current vulnerability, I simply forgot to address it. We also had some emergencies recently (developing features for our customers), that makes the security issues less prio than releasing the ordered features :-/ ... Anyway, our security approach is far from perfect, I am aware of it, and I'm willing to improve this, but sometimes it is difficult to explain our customers that it is a main point to consider in the development process.'*

In other developer feedback, we find developers perceive the practice of updating their dependencies as added effort and responsibilities that should be performed in their *'spare time'*. Moreover, developers suggest that availability of the manpower and assigned roles are influencing factors for deciding whether or not a developer will migrate a dependency:

– *'Just that it's not very easy to keep track of it. As there's no downside in upgrading in this case, we would have done so if for example there's a build time warning about such dependency.... Thank you very much for bringing this to our attention! We don't maintain XXX, but have passed this on to XXX. Not our decision, this is a slightly revised fork.'*
– *'I suppose we weren't aware of this issue.'* and that they have issues with the *'amount of people currently working on the project with spare time to verify it works correctly with new version'*.
– *'I can't answer for the group, but generally there are so many security vulnerabilities that it's a full time job just to keep up with them all. In most cases they don't apply.'*
– *'As mentioned above, we are no longer maintaining this particular version. Yes, but only "decentralized and informal one": whoever introduced the dependency is supposed to keep track of it and update it in the part(s) he is maintaining.'*

Based on these results, we return to answer (RQ3):

> We contacted developers to understand developer awareness and their opinions regarding the practice of updating their library dependencies. To answer (RQ3): *We find that 69% of developers were unaware of their vulnerable dependencies and proceeded to immediately migrate to a safer dependency. Developers evaluate the decision whether or not to update its dependencies based on project specific priorities. Developers cite migration as a practice that requires extra migration effort and added responsibility.*

7 Discussion

In this section, we discuss the implications of our results and the validity of our study.

7.1 Implications of Results

To understand implications of the study, in this section, we first discuss results of the research questions and speculate some factors that influences a developer to migrate a dependency. We then show how our work is relevant in the context of existing literature. Results for (RQ1) show that the heavy reliance on libraries in contemporary projects often results in the formation of complex inter-dependency relationships inside that project. Furthermore, responses in (RQ3) also provide evidence that the complex nature of these inter-dependencies (colloquially termed as *'dependency hell'*) is an influencing factor on whether or not a dependency will be migrated. When addressing (RQ2) and (RQ3), we find that migrating dependencies is a cost–benefit trade-off decision between estimating the amount of migration effort needed and evaluating the benefits of the replacement dependency. Finally, we speculate that developer workloads, responsibilities and a lack of motivation are important factors that influence whether or not a developer will migrate a dependency. Specifically, evidence from (RQ3) supports the notion that library migrations are perceived by developers as *'low priority'* and *'extra work to be done in their spare time'*.

We find our results to be consistent with existing literature that analyzed migrations at the API level of abstraction. At the fine-grained API level, related work defines migration effort as the effort needed to accommodate any changes in the API calls between a library and client system. Our study complements this migration effort from more coarse-grained level of abstraction, defining migration effort as the additional rework, testing and compatibility with other inter-dependencies when migrating a dependency. We speculate that developers are more likely to migrate a dependency if (i) they are aware of a migration opportunity, (ii) they have the time and responsibility to perform the migration effort, and (iii) they decide the migration aligns with the goals and objectives of their project. Our findings complement results in related work. For instance, Robbes et al. [2012] states that: *"A minority of reactions to API changes can remain undiscovered long after the original change is introduced"*. Furthermore, McDonnell et al. [2013] states that: *"Android APIs are evolving fast and client adoption is not catching up with the pace of API evolution"*. This result is further strengthened by the findings of Hora et al. [2015], who says that: *"...53% of the analyzed API changes caused reaction in 5% of the systems and affected 4.7% of developers"*. Our result complements these findings further by providing evidence that developers are more likely to be unaware of migration opportunities. A study by Bavota et al. [2015] comprehensively investigated API and library updates within the Apache ecosystem, with the goal of understanding what different product and process factors lead to developers updating their libraries. The study shows that at the API level, developers tend to upgrade a dependency when substantial changes such as bug-fixing activities are included in the replacement dependency. At a higher level of abstraction, we speculate that developer awareness of these bug-fixes and ease of integration determines whether or not the developer will migrate this dependency. Finally, to complement the findings of Bogart et al. [2015], our results reveal that non-technical organization factors (i.e., developer workload and responsibilities) play an important role in whether or not a developer will migrate their dependencies.

7.2 Threats to Validity

We now present construct, internal and external threats to our study.

Construct Validity - refers to the concern between the theory and achieved results of the study. We find three threats that relate to the tools and mechanisms used to obtain our results. The first is source of our datasets. In reality, there are other forms of awareness mechanism such as social media alerts or the word–of–mouth medium, to raise developer awareness to a migration opportunity. However, we believe that new releases and security advisories are more traditional and recognized forms of announcements. For future work, we plan to investigate other forms of awareness mechanisms that lead to a library migration. The second threat is the related to the tools used to extract our dependency migration information. In this study, we use the configuration file of the Maven dependency manager to assume the third-party dependencies. There may be cases were a third-party library is not declared in such configuration, and is instead embedded manually into the system. Our tool *PomWalker* cannot capture these dependencies as it specifically detects documented dependency declarations (i.e., implicit version references and managed dependencies). Our method also does not count dependencies that have copied the source code of the library into their own source code. However, since our collected dependencies under-estimate actual reuse, we do not believe this threat affects our main result. We also assume that explicit stated versioning is used by projects that are more likely to manage their dependencies. The final threat is the selection criteria used to select the case studies for (RQ2). The threat is that the criteria was performed manually (matching CVE to libraries) which could be error-prone and might have missed other case studies. However, we believe that our systematic research methods ensure a quality dataset and case studies that validate our drawn conclusions.

Internal Validity - refers to the concerns that are internal to the study. In this study, we found two main internal threats that could affect our results. First is the accuracy of our dataset and generalization of our results to represent the real world. This has an impact to both (RQ1) and (RQ2). For instance, a dataset containing obsolete projects or inactive forks of systems would cause false positives on the Library Migration Plot (LMP) trend curve. To mitigate this, we took particular care to filter out projects that were not regularly maintained by their developers. The second threat is related to the research method and actual response rate of the developers for RQ3. We understand that these little responses cannot be a true representation of all developers. However, we believe that the response rate of 25% from the smaller set of contactable projects is indeed adequate when targeting a specific interest group.

External Validity - refers to the generalization concerns of the study results. We have two main threats to the results of the study. First is the conclusions of the case studies to generalize the trend patterns for all Java projects.

Due to dataset quality pre-processing, we analyze projects that do use third-party libraries, hence, results may not be applicable to the other types of Java projects that do not use a dependency management tool. Also, there is a threat to the case studies not being representative of all projects. For new releases, we found that the three libraries depicted the typical pattern of either system users consistently migrating or where an older library version is the popular version. There may be other interesting patterns, but theoretically for new releases, these are two typical patterns for popular libraries. The second threat is the generalization to other library ecosystems such as JavaScript npm, Ruby RubyGems. We are careful to restrict these findings to Java projects as other ecosystems may depict a different set of library migration patterns and tendencies. This would be an interesting future avenues for research. We envision that different lessons learnt from other ecosystems in terms of responsiveness to library updates may provide insights on how to encourage library maintenance within the Maven ecosystem.

8 Related Work

Complementary to the related work of Robbes et al. [2012], Hora et al. [2015], McDonnell et al. [2013] and Bavota et al. [2015] already presented in the paper, there has been other work that have studies library migrations, both at the API and library component level. In this section, we cover the body of literature on library popularity, API library migrations and studies on software ecosystems.

API Library Updates - Teyton et al. [2014] studied library migrations of Java open source libraries from a set of client with a focus on library migration patterns. The main result of that study was that recommendations of libraries could be inferred from the analysis of the migration trends. In this work, we have a different motivation to how much migration occurs and especially in relation to vulnerabilities. Another work was by Xia et al. [2013], that studied the reuse of out-dated project written in the c-based programming language. Kabinna et al. [2016] and colleagues especially focused on the migration of specific logging libraries and not related to vulnerabilities.

Recently, large-scale empirical studies have been conducted on library updates. Raemaekers et al. [2012] performed several empirical studies on the Maven repositories about the relation between usage popularity and system properties such as size, stability and encapsulation. Raemaekers et al. [2014] also studied the relationship between semantic versioning and breakages. Other related empirical studies were conducted by Jezek et al. [2015] and Cox et al. [2015]. They studied in-depth how libraries that reside in the Maven Central super repository evolve. The motivation of our work differs from those work, as we are more focused on the migration process itself and its triggers rather than the migration effort needed to migrate a dependency.

Library Usage as popularity measures - The LU metrics and the LMPs are forms of popularity measure by the crowd. Popularity is not a new concept, with several research on usage trends of libraries. There has been work that has analyzed different dimensions on library usage by clients. For example, work such as De Roover et al. [2013] exploited library usage at the API level to understand popularity and usage patterns of clients. Similarly, they also looked at both the system and library dimensions of API usage for the Qualitas dataset of projects. The main differences to our work is that although overlapping, De Roover and colleagues analyzed at the API level, where we look at the higher abstraction of the library level. Moreover, instead of a simple popularity count, we define a model and metrics to quantify different metrics of LU.

Much like the LMP, related studies have used library usage visually to measure stability [McDonnell et al., 2013] or popularity [Mileva et al., 2009]. In this context, our previous work [Kula et al., 2014], among work leveraged popularity to recommend when libraries are deemed safe to use by the masses. Popularity has also been leveraged in IDEs. For instance, Eisenberg et al. [2010] improved navigation through a library's structure using the popularity of its elements to scale their depiction. Recently, Hora and Valente [2015] introduced their tool called *apiwave*, that visualizes popularity trends of a library at the API level.

Library migration support - There has been much research related to the transformation of client code to support library migration, particularly pertaining to the migration effort required. Work by Chow and Notkin [1996] and Balaban et al. [2005] used a change specification language. Wu et al. [2015b] showed in an empirical study that imperfect change rules can be used by developers upgrading their code, especially when documentation is lacking. There is work that provides the client automatic tool support to accommodate changes made to a library. For instance, SemDiff by Dagenais and Robillard [2009] recommended replacements for framework methods that were accessed by clients. Other similar tools were proposed by Xing and Stroulia [2007] and Schäfer et al. [2008]. In this work, we propose to view the migration from a higher level of abstraction at the library component level.

These tools also do not consider the other aspects of the migration process. Closely related to our work, Plate and Ponta [2015] stated that impact assessment, migration effort, and the customer are issues faced by the pragmatic developers wanting to update their vulnerable libraries. This study shows that these are indeed some of the reasons why maintainers are not updating, even in cases where they expose the software to outside malicious attacks.

Other work on reuse support is through code analysis. This area of work considers code clone detection techniques by Kamiya et al. [2002] to support which library version is most appropriate candidate for migration. Godfrey and Zou [2005] proposed origin analysis to recover context of code changes. Our previous work by Kawamitsu et al. [2014] tracked how code is reused across different code repositories. Also, work such as Cossette and Walker [2012] depict the complexities of the migration effort needed for library changes and transformations at the API level.

Software systems as ecosystems. Lungu [2008] best termed ecosystems as a 'collection of software projects which are developed and evolved together in the same environment'. The discussed work of Robbes et al. [2012], McDonnell et al. [2013] and Bavota et al. [2015] involved the analysis of API usage within a software ecosystem. Related, Wu et al. [2015a] explored the API changes and usages on Apache and Eclipse ecosystems. In this work, we also look at the Maven Java ecosystem of libraries, however, our clients are indeed from 'wild' real-world projects that reside in the much more diverse GitHub repository of repositories. More recent work has been by done by Wittern et al. [2016], who studied dynamics of the npm JavaScript library ecosystem.

Mens et al. [2014] performed ecological studies of the R CRAN open source software ecosystems. Haenni et al. [2013] performed a survey to identify the information that developers lack to make decisions about the selection, adoption and co-evolution of upstream and downstream projects in a software ecosystem. Similar works were performed by German et al. [2013] for the R software ecosystem. The external library dependencies could be considered as part of the ecosystem. Therefore, the larger ecosystem of library dependencies may also trigger migrations. However, in this study, we focused on the trigger effect of vulnerabilities and updates within the same library.

9 Conclusion

Many software projects today advocate the use of third-party libraries because of its many benefits for software developers. However, results of this study show that updates of third-party library dependencies are not regularly practiced, especially to fix vulnerabilities that exploit a system to attackers. Surprisingly, we found that 81.5% of our studied systems still remain with an outdated dependency. The study shows many factors that influence the decision whether or not to update a library. Migration effort such as rework required prepare a system to work on a new platform (i.e., Java 4 to Java 5) and address the API changes plays an important role in the update decision. Developer awareness also influences the migration process and they do not prioritize updates by questioning the migration cost, citing it as added responsibility and effort to be performed in their *'spare time'*. We speculate other issues including developer responsibilities and a lack of motivation play a role in the decision on whether a dependency will be migrated or not.

The study provides motivation for our community develop strategies to improve a developer personal perception of third-party updates, especially in cases when effort must be allocated to mitigate a severe vulnerability risk. Visual aids such as the Library Migration Plots (LMP) provide a rich visual analysis, which proves to be a useful awareness and motivation for developers to identify dependency migration opportunities. We envision this work as a contribution toward developing strategies and support tools that aid the management of third-party dependencies.

Acknowledgements This work is supported by JSPS KANENHI (Grant Numbers JP25220003 and JP26280021) and the "Osaka University Program for Promoting International Joint Research."

References

Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 265–279, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0.

Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. How the apache community upgrades dependencies: An evolutionary study. *Empirical Softw. Eng.*, 20(5):1275–1317, October 2015. ISSN 1382-3256.

Christopher Bogart, Christian Kästner, and James Herbsleb. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *Proceedings of the ASE Workshop on Software Support for Collaborative and Global Software Engineering (SCGSE)*, 11 2015.

Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *Proceedings of the 1996 International Conference on Software Maintenance*, ICSM '96, Washington, DC, USA, 1996. IEEE Computer Society.

Bradley E. Cossette and Robert J. Walker. Seeking the ground truth. *Proc. of the ACM SIGSOFT Intrn. Symp. on the Foundations of Software Engineering - FSE '12*, 2012.

Joel Cox, Eric Bouwers, Marko van Eekelen, and Joost Visser. Measuring dependency freshness in software systems. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 2, pages 109–118, May 2015.

Barthelemy Dagenais and Martin P. Robillard. Semdiff: Analysis and recommendation support for api evolution. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 599–602, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4.

Coen De Roover, Ralf Lammel, and Ekaterina Pek. Multi-dimensional exploration of API usage. *IEEE International Conference on Program Comprehension*, pages 152–161, 2013.

S. Edgell and S. Noon. Effect of violation of normality on the t test of the correlation coefficient. In *Psychological Bulletin*, pages 576–583, 1984.

Daniel S. Eisenberg, Jeffrey Stylos, Andrew Faulring, and Brad A. Myers. Using association metrics to help users navigate API documentation. In *VL/HCC2010*, pages 23–30, 2010.

Daniel M. German, Bram Adams, and Ahmed E. Hassan. The evolution of the r software ecosystem. *Proc. of European Conf. on Soft. Main. and Reeng. (CSMR2013)*, pages 243–252, 2013.

M.W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, February 2005.

Nicole Haenni, Mircea Lungu, Niko Schwarz, and Oscar Nierstrasz. Categorizing developer information needs in software ecosystems. In *Proc. of Int. Work. on Soft. Eco. Arch. (WEA13)*, pages 1–5, 2013.

André Hora and Marco Tulio Valente. apiwave: Keeping track of api popularity and migration. In *International Conference on Software Maintenance and Evolution*, 2015.

Andre Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stephane Ducasse, and Marco Tulio Valente. How do developers react to api evolution? the pharo ecosystem case. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ICSME '15, pages 251–260, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4673-7532-0. doi: 10.1109/ICSM.2015.7332471.

Kamil Jezek, Jens Dietrich, and Premek Brada. How Java APIs break - An empirical study. *Information and Software Technology*, pages 129–146, 2015. ISSN 09505849. doi: 10.1016/j.infsof.2015.02.014.

Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E. Hassan. Logging library migrations: A case study for the apache software foundation projects. In *Proceedings of the 13th International Workshop on Mining Software Repositories*, MSR '16, pages 154–164, New York, NY, USA, 2016.

T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002. ISSN 0098-5589. doi: 10.1109/TSE.2002.1019480.

Naohiro Kawamitsu, Takashi Ishio, Tetsuya Kanda, Raula Gaikovina Kula, Coen De Roover, and Katsuro Inoue. Identifying source code reuse across repositories using lcs-based source code similarity. In *Proc. of SCAM*, 2014.

Raula G. Kula, Coen D. Roover, Daniel M. German, Takashi Ishio, and Katsuro Inoue. Visualizing the evolution of systems and their library dependencies. In *Proc. of IEEE Work. Conf. on Soft. Viz. (VISSOFT)*, ICSME '15, 2014.

Raula Gaikovina Kula, Daniel M. German, Takashi Ishio, and Katsuro Inoue. Trusting a library: A study of the latency to adopt the latest maven release. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, Canada, March 2-6, 2015*, 2015.

M. M. Lehman. Laws of software evolution revisited. In *Proceedings of the 5th European Workshop on Software Process Technology*, EWSPT '96, pages 108–124, London, UK, UK, 1996. Springer-Verlag. ISBN 3-540-61771-X.

Mircea Lungu. Towards reverse engineering software ecosystems. In *Intl. Conf. on Soft. Maint. and Evo. (ICSME)*, 2008.

Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of API stability and adoption in the android ecosystem. *IEEE International Conference on Software Maintenance, ICSM*, pages 70–79, 2013. ISSN 1063-

6773. doi: 10.1109/ICSM.2013.18.

Tom Mens, Maelick Claes, and Philippe Grosjean. Ecos: Ecological studies of open source software ecosystems. In *Soft. Main. Reeng. and Rev. Eng. (CSMR-WCRE)*, pages 403–406, Feb 2014.

Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. Mining trends of library usage. In *Proc. Intl and ERCIM Principles of Soft. Evol. (IWPSE) and Soft. Evol. (Evol) Workshops*, IWPSE-Evol '09, pages 57–62, New York, NY, USA, 2009. ACM.

Henrik Plate and Sabetta Antonino Ponta, Serena Elisa. Impact assessment for vulnerabilities in open-source software libraries. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution*, ICSME '15, Breman, Germany, 2015. IEEE Computer Society.

S. Raemaekers, A. van Deursen, and J. Visser. Measuring software library stability through historical version analysis. In *Proc. of Intl. Comf. Soft. Main. (ICSM)*, pages 378–387, Sept 2012.

S. Raemaekers, A. van Deursen, and J. Visser. Semantic versioning versus breaking changes: A study of the maven repository. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 215–224, Sept 2014.

Romain Robbes, Mircea Lungu, and David Röthlisberger. How do developers react to api deprecation?: The case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 56:1–56:11, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1614-9.

Everett M. Rogers. *Diffusion of innovations*. Free Press, NY, 5th edition, 08 2003. ISBN 0-7432-2209-1, 978-0-7432-2209-9.

Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. On the reaction to deprecation of 25,357 clients of 4+1 popular java apis. In *Proceedings of the 32th IEEE International Conference on Software Maintenance and Evolution.*, 2016.

Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining framework usage changes from instantiation code. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 471–480, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1.

Cdric Teyton, Jean-Rmy Falleri, Marc Palyart, and Xavier Blanc. A study of library migrations in java. *Journal of Software: Evolution and Process*, 26 (11), 2014.

Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the javascript package ecosystem. In *Proc. of Work. Conf. on Mining Soft. Repo. (MSR2016)*, 2016.

Wei Wu, Foutse Khomh, Bram Adams, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of api changes and usages based on apache and eclipse ecosystems. *Empirical Software Engineering*, pages 1–47, 2015a. ISSN 1573-7616. doi: 10.1007/s10664-015-9411-7.

Wei Wu, Adrien Serveaux, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. The impact of imperfect change rules on framework api evolution identification:

An empirical study. *Empirical Softw. Engg.*, 20(4):1126–1158, August 2015b. ISSN 1382-3256. doi: 10.1007/s10664-014-9317-9.

Pei Xia, Makoto Matsushita, Norihiro Yoshida, and Katsuro Inoue. Studying reuse of out-dated third-party code in open source projects. In *Japan society for Software Science and Technology*, Computer Software, Vol.30, No.4, pp.98-104, 2013.

Zhenchang Xing and Eleni Stroulia. API-evolution support with diff-catchup. *Software Engineering, IEEE Transactions on*, 33:818–836, 2007. doi: 10.1109/TSE.2007.70747.