Trusting a Library: A Study of the Latency to Adopt the Latest Maven Release

Raula Gaikovina Kula*, Daniel M. German*‡ Takashi Ishio*, Katsuro Inoue*
* Osaka University, Japan ‡ University of Victoria, Canada
{raula, ishio, inoue}@ist.osaka-u.ac.jp
dmg@turingmachine.org

***Abstract*—With the popularity of open source library (re)use in both industrial and open source settings, *'trust'* plays vital role in third-party library adoption. Trust involves the assumption of both functional and non-functional correctness. Even with the aid of dependency management build tools such as Maven and Gradle, research have still found a latency to trust the latest release of a library. In this paper, we investigate the trust of OSS libraries. Our study of 6,374 systems in Maven Super Repository suggests that 82% of systems are more trusting of adopting the latest library release to existing systems. We uncover the impact of maven on latent and trusted library adoptions.**

I. Introduction

As software systems grow, so does its size and complexity. Known software engineering best practices suggest that we re-compose code into easy–to–understand and maintainable functional modules or libraries. With the emergence of large collections of software repositories over the Internet, now systems integrate popular Open Source Software (OSS) libraries like JUNIT[1] and SPRING[2]. The effect of heartbleed bug[3] and more recently the shellshock vunerability[4] demostrates OSS library widespread effect on everyday systems. Online collections of libraries together form *super repositories* like GitHub[5] and the Maven Central Repository[6].

Well-known benefits of adopting libraries includes efficient development with credible assurance of quality. *'Trust'* of a library plays an important role. Trust involves the assumption of a module's functional and non-functional correctness. For instance, system maintainers need to trust the reliability of non-functional attributes such as security and stability of an adopted library.

Libraries, much like any software, are never perfect. Over time, release updates are sometimes necessary to address bugs or perform other maintenance activities such as refactoring. New features are added to expand functionality. Some updates become mandatory and is crucial patch vulnerabilities. However, sometimes updates break systems, causing incompatibility issues with shared system-wide resources. According to maven, *'dependency management becomes crucial, especially with large-scale multi-modules software systems with tens of hundreds of modules'*.[7] The tracking and maintenance of libraries has come to be known as dependency management issues (colloquially termed as *'dependency hell'*[1]). To handle dependency management issues, many system maintainers utilize build tools such as Maven[8] and Gradle[9]. These tools promote updates and the trust of latest available library releases in the super repository.

However, studies by Raemaekers *et. al.*[2], [3] and our previous work [4], [5] all report evidence of latency when updating to the latest release of a library. Incompatibilities, such as API breakages and technical rework needed to facilitate the new library may cause such delays. Other speculations involve the lack of awareness of a newer library release. Since trust has a human aspect, other various reasons may exist.

In this study, we investigate the impact of dependency management tools such as maven have on trusting a library. We coin the term *'trusted adoption'* to when the latest release of new library is adopted. Conversely, *'latent adoption'* refers to when a maintainer does not adopt the latest library release.

The motivation is to investigate the latency of trust phenomena. We carried out an empirical study to evaluate trusted adoption of OSS libraries within the maven super repository. The following research questions guided our study:

1) *How much 'latent adoption' exists?* In RQ1, we empirically detect when a system does not adopt the latest release of a new library.
2) *What is the current trend of maintainers trust?* In RQ2, we investigate the trend over recent times to understand the current state of trust of OSS libraries.

We present a set of algorithms to classify initial and introduced library dependency types. Our results suggest that with the current tools and technology, latent adoption is still common. Maintainers, however, are more trusting when introducing new libraries into existing systems.

II. Background and Motivation

A. Trust of a Library

We consider trust to play an important role when choosing and maintaining a third-party library. Trust intrinsically is a human trait, thus making it very difficult to 'pin down' to a

[1]junit.org
[2]spring.io
[3]http://heartbleed.com/
[4]https://shellshocker.net/
[5]https://github.com/
[6]http://search.maven.org/
[7]http://goo.gl/TO2P1u
[8]http://maven.apache.org/index.html
[9]http://www.gradle.org/

978-1-4799-8468-8/15/$31.00 © 2015 IEEE

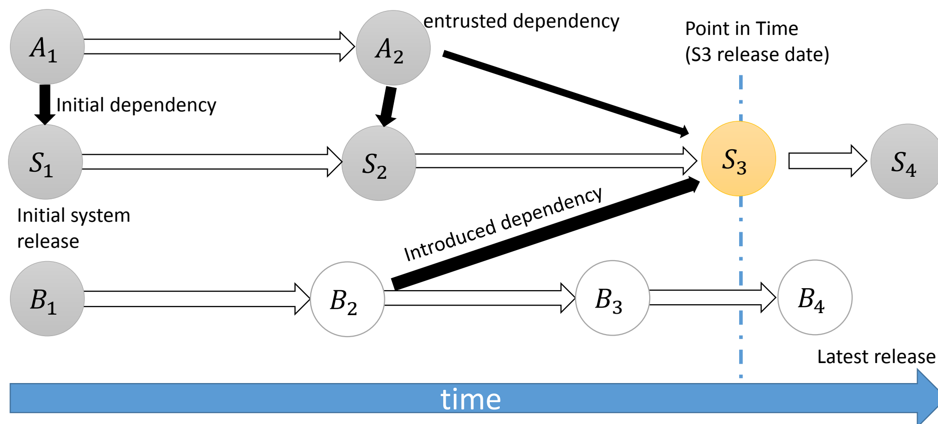

Fig. 1: Example of systems and library dependencies over time. The Figure depicts the evolution of system S with libraries A and B.

generalized reason. Trust is at varying degrees and for many reasons. In this paper we consider four broad types of trust that would cause a latent adoption.

The first type of trust is of a library's both functional and non-functional related specifications. Examples of non-functional correctness are missing or inconsistency of functions corresponding to the API documentation. Similarly, trusting the correct use of Semantic Versioning (SemVer - MAJOR.MINOR.PATCH)[10] matching the changes in the release (i.e, major changes are only introduced in major version releases). A study by Raemaekers [6] uncovered that many libraries do not follow the SemVer schema, causing many breaking changes which could lead to latent adoptions.

The second type of trust is that the introduced library is not volatile towards the current system environment. This involves compatibility with other libraries or the shared-system resources. Take for example the java ASM[11] library. ASM is a tiny and flexible java byte-code manipulation library used in many popular frameworks like HIBERNATE[12], SPRING, GLASSFISH[13] and DERBY SERVER[14]. It is known for its fast execution speed due to its small size. The cost of keeping such a small size was that incompatibilities would occur whenever a new JDK compiler was released. The main issue occurred when systems adopted libraries that internally had different versions of ASM as they would break. For instance, systems that had adopted the DERBY server and SPRING framework encountered incompatibility issues when Spring updated from $\text{ASM}_{3.x}$ to $\text{ASM}_{4.x}$. Eventually latter releases of ASM allowed for backward compatibility. This example illustrates the importance of the ASM library development team influence on trust adoptions[15].

We refer to the third type of trust as loyalty for a certain version of a library based on experienced usage. For instance, in an informal interview, a developer replied that: *'although it is an way older version, I use Junit$_{3.8}$ as I am familiar with it. It is not a critical compile component of my system and suits my needs.'* This trust of a particular release promotes latent adoption.

Finally, a system maintainer may not trust the test maturity of latest version release. In informal interviews, we found that many maintainers are hesitant of the rigor of internal testing. They believe that undetected bugs that can only be uncover during in practical use may exist. In this case, maintainers search for most stable version (in many cases deemed as the popular) release.

[10]http://semver.org/
[11]http://asm.ow2.org/index.html
[12]http://hibernate.org/
[13]https://glassfish.java.net/
[14]http://db.apache.org/derby/
[15]This is the report of the asm compatibility issues http://goo.gl/x9EeKx known as the asm NoSuchMethodError incompatibility issue

III. SOFTWARE SYSTEMS AND LIBRARIES

We investigate the evolving relationship between systems and their dependent libraries. In this research we are only concerned with the adoption of new libraries. Updates to existing library dependencies are ignored. We used the model in Figure 1 to describe the necessary terminology.

A. Terminology and Definitions

- **System and Library Releases.** We conjecture that systems form a dependency relationship with a set of libraries. A *release* refers to an instance of a system or library. Let S_{ver} be the notation for a unique release of a software system and L_{ver} as libraries. In Figure 1, S_1, S_2, S_3, S_4 are different system releases of system S. A_1, A_2, are releases of library A and B_1, B_2, B_3, B_4 as library B releases. As depicted in Figure 1, all related releases are connected.
- **Releases Ordering.** Since releases have a temporal property, we can order releases by their release date. Thus, $pre(S_{ver})$ refers to a previous release while $suc(S_{ver})$ refers to successive releases of a system. Likewise for $pre(L_{ver})$ and $suc(L_{ver})$.
We use the notation $>$ and $<$ to represent the temporal ordering between all system and library releases. For instance, in Figure 1, $B_4 > S_3 > B_3$ describes the ordering between B_4, S_3 and B_3. We use the ordering

to identify particular releases. The *initial* release is used to describe an instance where no predecessor exists. (i.e $\nexists pre(S_{ver})$ or $\nexists pre(L_{ver})$). For example, in Figure 1, releases A_1, S_1 and B_1 are initial releases. The *latest* release is such that no successive release exists (i.e $\nexists suc(S_{ver})$ or $\nexists suc(L_{ver})$). In Figure 1, the latest releases are A_2, S_4 and B_4 respectively.

- **Dependency Relations.** This refers to the dependency relationship that forms when a library is being adopted by a system. Suppose a system S_x and library L_y, $depends(S_{ver}, L_{ver})$ describes to the adoption of library L_y into a system S_x. In Figure 1, for system S, $depends(S_1, A_1)$, $depends(S_2, A_2)$ $depends(S_3, A_2)$ and $depends(S_3, B_2)$ depicts how libraries A and B are adopted. As shown in Figure 1, dependency relations are the edges that exist between a system and library (i.e, edge between A_1 and S_1).
- **Initial Dependency.** An initial dependency is all dependencies that exist at the initial system release. (i.e $depends(S_{ver}, L_{ver})$ where $\nexists pre(S_{ver})$ conditions must be met). As depicted in Figure 1, $depends(S_1, A_1)$ is an initial dependency for system S.
- **Introduced Dependency.** An introduced dependency refers to new dependencies introduce to an already existing system. (i.e $depends(S_{ver}, L_{ver})$ where $\exists pre(S_{ver})$ and $\nexists depends(pre(S_{ver}), L_{ver})$ conditions must be met). In Figure 1, $depends(S_3, B_2)$ is an example of an introduced dependency.
- **Trusted Dependency.** Trusted dependency is the adoption of the closest library release at a particular point in time. (i.e $depends(S_{ver}, L_{ver})$). One of the two conditions must be met 1.) L_{ver} is the latest version such that $\nexists suc(L_{ver})$ and 2.) If condition 1 is not true, (i.e, $\exists suc(L_{ver})$), then $suc(L_{ver})$ must be released after the system release S_{ver} (i.e, $suc(L_{ver}) > S_{ver}$).

In Figure 1, taking S_3 as a reference point, the dependency $depends(S_3, B_2)$ is not an trusted dependency adoption. This is because B_2 is not the latest release. and the successor B_3 was released before S_3 ($B_3 < S_3$). The entrusted dependency would be $depends(S_3, B_3)$ as the successive release B_4 was released after S_3 (i.e, $B_4 > S_3$). Also, in Figure 1, $depends(S_3, A_2)$ is an trusted dependency as A_2 is the latest release of library A (condition 1 is satisfied).

B. Dependency Algorithms

To determine a trusted dependency, we follow three steps as depicted in Figure 2. Suppose for a system S and n represents all of the system releases of S, we perform the following steps:

1) *Extract related dependency relations*. For all releases of S (i.e, $S_1, ..., S_n$), we extract all related dependency relations.
2) *Determine dependency relation*. In this step we distinguish if the dependency is either an initial or introduced dependency.
3) *Determine trusted dependency*. To determine a trusted introduced dependency, we check the conditions to satisfy a trusted dependency.

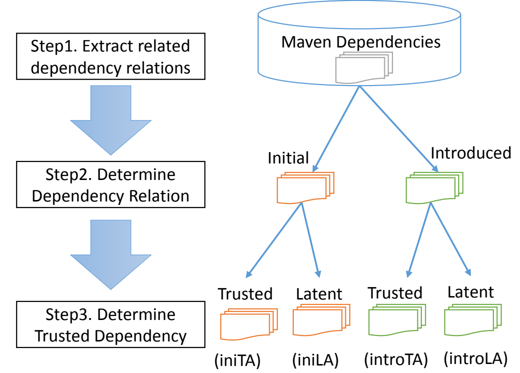

Fig. 2: The Figure depicts the algorithms used to classify our proposed types of dependency adoptions.

As shown in Figure 2 using the algorithm we classify into the interested dependencies. Formally each dependency is defined as follows:

a) Initial Trusted Adoption (iniTA): Trusted adoption of latest library release at initial conception of a system.

$$iniTA(S) = \{\bigcup_{i \in n} \exists x.depends(S_i, L_x) | \nexists pre(S_x) \\ \wedge (\nexists suc(L_x) \vee if(\exists suc(L_x))then(suc(L_x) > S_i))\} \quad (1)$$

b) Initial Latent Adoption (iniLA): Latent adoption at initial conception of a system.

$$iniLA(S) = \{\bigcup_{i \in n} \exists x.depends(S_i, L_x) | \\ \nexists pre(S_x) \wedge (\exists suc(L_x) \wedge (suc(L_x) > S_i))\} \quad (2)$$

c) Introduced Trusted Adoption (introTA): Trusted adoption of latest library release introduced to an existing system.

$$introTA(S) = \{\bigcup_{i \in n} \exists x.depends(S_i, L_x) | \nexists pre(L_x) \\ \wedge (\nexists suc(L_x) \vee if(\exists suc(L_x))then(suc(L_x) > S_i))\} \quad (3)$$

d) Introduced Latent Adoption (introLA): Latent adoption of library introduced to an existing system.

$$introLA(S) = \{\bigcup_{i \in n} \exists x.depends(S_i, L_x) | \nexists pre(L_x) \\ \wedge (\exists suc(L_x) \wedge (suc(L_x) > S_i))\} \quad (4)$$

IV. Empirical Study

We implemented our trust dependency algorithms to study the 'latent adoption' phenomena between maven 2 repository libraries. The Maven Super Repository is the home to a large collection of software libraries that originate from the Java Virtual Machine (JVM) based languages such as Java, Scala and Clojure. Systems in the maven repository includes a Project Object Model file `POM.xml` that describes the project's configuration meta-data —including its compile-time and run-time dependencies. We developed a tool to extract this dependency information from all versions of the POM-files in

TABLE I: Maven Super Repository

	Maven Dataset
Time Period	2005-11-03 to 2013-11-24
# Dependency Relations	188,951
# Implicit Relations	11, 195
# of Systems	6,374
# of Libraries	5,146

TABLE II: Dependency Classification Results

	# Libraries	# Dependencies (%)
iniTA	4,192	20,372 (59.63%)
iniLA	848	13,791 (40.37%)
introTA	3,064	29,303 (81.16%)
introLA	823	6,543 (18.24%)

the repository (PomWalker[16]). Our implementation relies on `version 3.1.1` of maven's own `maven-model` project to parse each POM file and extract syntactic references to its dependencies.

POM files can either reference be explicit to a specific release version or be implicit. For instance, the syntax (`<version>$library.version</version>`) is used to import managed dependencies. For simplicity, we only used explicit release versions in our study. We implemented R scripts of the dependency algorithms. Using the extracted dependency information, we were then able to determine trusted dependencies. All scripts and tools are available as a replication package[17].

V. Results

Table I is a summary of the data collected by PomWalker. We extracted 188,951 dependency relations from the maven super repository. This consists of 6,374 systems using 5,146 libraries over 7 years. Table II refers to the extracted introduced and initial entrusted dependencies. In the case of initial dependencies, 59.63% of the detected dependencies trusted the latest release, while 40.37% did not adopt the latest release version. The results show a relatively smaller set of latent

[16]https://github.com/raux/PomWalker

[17]http://sel.ist.osaka-u.ac.jp/~raula-k/LatentAdoption/

TABLE III: Top 5 Initial Trusted Libraries

Library	# Dependencies
scala-library	710
junit	595
log4j	191
slf4j-api	178
commons-logging	169

TABLE IV: Top 5 Initial Latent Libraries

Library	# Dependencies
scala-library	1,468
junit	1,411
servlet-api	496
commons-logging	481
log4j	394

TABLE V: Top 5 Introduced Trusted Libraries

Library	# Dependencies
junit	243
hibernate-core	238
spring-integration-core	235
scala-library	227
hibernate-testing	217

TABLE VI: Top 5 Introduced Latent Libraries

Library	# Dependencies
junit	392
glassfish-corba-orbgeneric	126
glassfish-corba-omgapi	122
httpclient	120
commons-logging	119

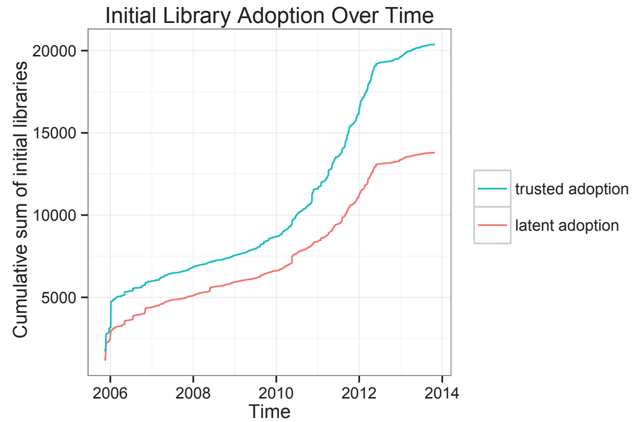

Fig. 3: A comparision of trust of initial dependencies over time in the Maven Super Repository.

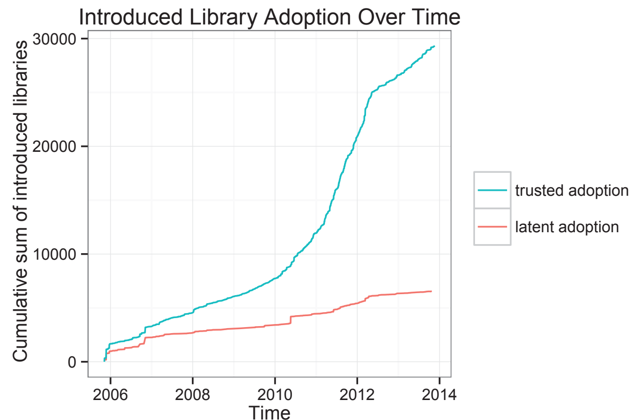

Fig. 4: A comparision of trust of introduced dependencies over time in the Maven Super Repository.

adopted libraries (848) compared to their trusted counterparts (4,192). In the case of introduced libraries, 81.16% of the adopted libraries were latest releases. The results reveals a relatively smaller set of latent adopted libraries (823) that maintainers introduced into their systems.

The Top 5 libraries adopted at the conception of a system are shown in Table III (intital trust adoption) and Table IV (initial latent adoption). For introduced libraries, Table V (introduced trusted adoption) and Table VI (introduced latent adoption). Notice that popular libraries like SCALA-LIBRARY and JUNIT are apparent in all the dependency types. Both Figure 3 and Figure 4 depicts a cumulative number of initial and introduced library adoptions in the maven super repository from 2006 to 2014. Figure 3 shows a more steady trend between trusted and latent adoptions. On the other hand, in Figure 4 it is observed that when introducing new libraries, systems are more trusting of the latest releases.

VI. Implications of Results

Issues of updating libraries stems from the trust of releases. We did not study if these libraries eventually update these libraries, but a latent adoption may set a precedence of not updating libraries. The study shows a smaller subset of libraries may be responsible for latency in adoptions. Future work is towards identification and exploration of approaches to reduce latent adoptions. The study finds that trust adoptions are more apparent with introduced libraries. We speculate that introducing new libraries could indicate of adding new features to a system.

We are only beginning to understand the trust involved with OSS libraries. More study into trust of libraries will lead to better strategies to support component-based software. For example, how to 'certify' trust of a newly released release. Now we answer our research questions:

RQ1. *How much 'latent adoption' exists?:* The results show that the latent adoption exists, particularly at the initial conception of a system. Maintainers are less likely to adopt the latest releases at the beginning of project.

RQ2. *What is the current trend of maintainers trust?:* Figure 3 and Figure 4 suggests that maven libraries are becoming more inclined to adopt the latest releases when introducing new libraries (updating there existing systems).

VII. Related Work

The trust of components is well-known in fields of Dependable and Secure Computing [7], [8], [9]. In this paper, we explore how trust of components can be realized with OSS libraries. Related work in Software Engineering are concerned with library API usage and popularity. Holmes *et al.* appeal to popularity as the main indicator to identify libraries of interest [10]. De Roover *et al.* explored library popularity in terms of source-level usage patterns [11]. Mileva *et al.* study popularity over time to identify the most commonly used library versions [12]. These studies to an extent, are evidence to update to the most popular as opposed to the latest release of a library. More recently, work by Teyton *et al.* [13] and Cossette *et al.* [14] have explored library migrations, particularly on candidate library replacements. Both studies provide additional insights behind latent library adoptions.

VIII. Conclusion and Future Work

In this paper we explore the notion of trust when re(using) OSS libraries. Our results suggest that maintainers are becoming more trusting. Existing systems more inclined to adopt the latest releases to existing systems. The reasons for latent adoption are difficult to generalize. Related work reports lapses in updates and SemVer guidelines not being strictly enforced. Therefore, we believe that further investigation behind the reasons trust issues should be performed, such as maintainers satisfaction surveys on our current tools. We envision that future dependency management tools like maven should ensure OSS libraries are trustworthy, possibly resolving our dependency updating issues.

Acknowledgment

This work is supported by projects 'Collecting, Analyzing, and Evaluating Software Assets for Effective Reuse', Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (No.25220003) and 'Software License Evolution Analysis', Osaka University Program for Promoting International Joint Research.

References

[1] M. Jang, "Linux annoyances for geeks," 2009.
[2] S. Raemaekers, A. van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," in *Proc. of Intl. Comf. Soft. Main. (ICSM)*, Sept 2012, pp. 378–387.
[3] S. Raemaekers, G. Nane, A. van Deursen, and J. Visser, "Testing principles, current practices, and effects of change localization," in *Mining Soft. Repo. (MSR)*, May 2013, pp. 257–266.
[4] R. G. Kula, C. D. Roover, D. M. German, T. Ishio, and K. Inoue, "Visualizing the evolution of systems and their library dependencies," *Proc. of IEEE Work. Conf. on Soft. Viz. (VISSOFT)*, 2014.
[5] N. Kawamitsu, T. Ishio, T. Kanda, R. G. Kula, C. D. Roover, and K. Inoue, "Identifying source code reuse across repositories using lcs-based source code similarity," in *Proc. of SCAM*, 2014.
[6] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning versus breaking changes: A study of the maven repository," in *Proc. of SCAM*, Sept 2014, pp. 215–224.
[7] Z. Yan and C. Prehofer, "Autonomic trust management for a component-based software system," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 6, pp. 810–823, 2011.
[8] W. Hasselbring and R. Reussner, "Toward trustworthy software systems," *Computer*, vol. 39, no. 4, pp. 91–92, April 2006.
[9] R. V. Guha, R. Kumar, P. Raghavan, and A. Tomkins, "Propagation of trust and distrust," in *WWW'04*, pp. 403–412.
[10] R. Holmes and R. J. Walker, "Informing Eclipse API production and consumption," in *OOPSLA2007*, 2007, pp. 70–74.
[11] C. De Roover, R. Lämmel, and E. Pek, "Multi-dimensional exploration of api usage," in *Proc. of Int. Conf. on Prog. Comp.(ICPC)*, 2013.
[12] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "Mining trends of library usage," in *ERCIM Workshops*, 2009, pp. 57–62.
[13] C. Teyton, J.-R. Falleri, M. Palyart, and X. Blanc, "A study of library migrations in java," *Journal of Software: Evolution and Process*, vol. 26, no. 11, 2014.
[14] B. E. Cossette and R. J. Walker, "Seeking the ground truth: a retroactive study on the evolution and migration of software libraries," in *Proceedings of ACM SIGSOFT Int. Symp. on the Found. of Soft. Eng. (FSE)*, 2012.