Florida Institute of Technology
Scholarship Repository @ Florida Tech

Theses and Dissertations

5-2020

Dynamics in Recommendations of Updates for Free Open-Source Software

Shakre Elmane

Follow this and additional works at: https://repository.fit.edu/etd

Part of the Computer Sciences Commons

Dynamics in Recommendations of Updates for Free Open-Source Software

by
Shakre Elmane

Master of Science
Computer Science
Florida Institute of Technology
2020

A dissertation submitted to
the Department of Computer Engineering and Sciences of
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
in
Computer Science

Melbourne, Florida
May, 2020

© Copyright 2020 Shakre Elmane
All Rights Reserved

The author grants permission to make single copies

We the undersigned committee hereby recommend that the attached document be accepted as fulfilling in part the requirements for the degree of Ph.D. in Computer Science.

"Dynamics in Recommendations of Updates for Free and Open-Source Software"
a dissertation by Shakre Elmane

Marius C. Silaghi, Ph.D.
Associate Professor, Computer Science
Major Advisor

William H. Allen, Ph.D.
Associate Professor, Computer Science
Committee Member

Debasis Mitra, Ph.D.
Professor, Computer Science
Committee Member

Veton Kepuska, Ph.D.
Associate Professor, Computer Engineering
Committee Member

Philip Bernhard, Ph.D.
Associate Professor and Head
Department of Computer Engineering and Sciences

ABSTRACT

Dynamics in Recommendations of Updates for Free and Open-Source Software

by

Shakre Elmane

Thesis Advisor: Marius C. Silaghi, Ph.D.

In Free and Open-Source Software (*FOSS*) world, newer is not always better. Automatically updating to the latest version of FOSS applications involves real risks. The newer version could be missing features that are essential to some users, but are dropped by the developers. Another possible scenario, with even more serious consequences, is a project taken over by malicious developers who target users' sensitive data, or try to control their systems.

In this work we identify a set of security risks associated with changes of reviewers in automatic Free and Open-Source software (FOSS) updates. Automatic updates can be a prime target for attackers. Attackers that can compromise this process get access to a large number of users' machines. In addition to issues associated with regular software updates, automatic FOSS updates face more challenges ranging from developers dropping support for some features that are considered essential for some users, to malicious developers taking over a project, and providing versions of the software containing back-doors

to sensitive users' data. The lack of contracts between FOSS developers and the end users of their products allows for significant changes in quality and functionality of the produced software. Another issue that is unique to FOSS development is the possibility of having multiple competing branches of the same software being developed by different teams, as in the case of *Linux* distributions.

Stacking the Deck Attack is an example of an attack on automatic FOSS updates, when malicious agents control the development of a project, and purposely remove features to disrupt important processes, such as voting.

Existing solutions to address these challenges include utilizing *meta-recommenders* to rank the independent *reviewers*. These reviewers evaluate and recommend software updates and distributions. More diverse and stable recommenders boost the robustness against a take-over. We observe issues caused by reviewers churning (reviewers joining or leaving the system). We show that outdated recommendations from reviewers that are no longer active can continue to be maintained in the system and compete with active reviewers. We discuss here the implications of reviewer churning and we suggest and analyze solutions to mitigate these issues.

In this research, we improve on the existing *FOSS Updates Meta-Recommendations* framework, which is shown (in [6, 7]) to increase resistance to certain attacks. We study how to handle more efficiently the situations where reviewers join and/or leave the P2P network, without a significant impact on the accuracy of recommendations or the performance of the system. Here, algorithms for countering Stacking-the-Deck Attacks in the context of

reviewer churning are proposed based on distributed meta-recommenders, and are shown to help mitigate reasonable scenarios of attack & churning intensity, with limited casualty rates.

Table of Contents

Abstract iii

List of Figures ix

List of Tables xi

Acknowledgments xii

Dedication xiii

Chapter 1 Introduction 1

1.1 Meta Recommenders 6

1.2 Privacy in Distributed Constraint Satisfaction Problems (DisCSPs) 7

1.3 Contribution Summary 7

1.4 Motivation 8

1.5 Future Work 9

1.6 Chapters of the Dissertation 9

Chapter 2 Background and Literature Review **11**

2.1 Free and Open-Source Software (FOSS) 11

2.2 Updates in Free and Open-Source Software 14

2.3 Attacks in Software Updates 16

2.4 Recommender Systems 19

2.4.1 Recommender Systems Uses 21

2.4.2 Recommender Systems Types 22

2.5 Meta-Recommendations Framework (MRF) for FOSS Updates 25

2.5.1 Trust Propagation in the Meta-Recommendations Framework . . . 26

2.5.2 Meta-Recommendations Framework (MRF) Concepts 28

2.5.3 The Meta-Recommendations Framework Roles 31

2.5.4 Privacy in Multi-Agent Systems 32

Chapter 3 Concepts **36**

3.1 Effects Due to Reviewer Churning 37

3.2 Handling Reviewer Churning 38

3.3 Concepts 41

Chapter 4 Algorithms **43**

4.1 Choosing a Software Release 43

4.2 Updating Agent's Reviewers List 45

4.2.1 Reviewers Joining the System 47

4.2.2 Reviewers Leaving the System 48

4.3 Algorithms to Handle Churning 49

4.3.1 Drop/remove Messages 50

4.3.2 Limited-Life Meta-Recommendations Algorithm (LLMRA) 51

4.3.3 Non-positive Meta-Recommendations Algorithm (NPMRA) 54

4.3.4 Update-driven Meta-Recommendations (UDMRA): 54

Chapter 5 Experimentation **60**

5.1 Computational Environment 61

5.2 Simulator 61

5.3 Problem Generator 61

5.4 Parameters 61

5.5 Experiments 63

Chapter 6 Conclusions **79**

References **82**

List of Figures

2.1 Centralized FOSS development model 12

2.2 Decentralized FOSS updates framework 25

2.3 Messages exchanged between peers 28

2.4 BLA vs. MRF 32

2.5 BLA: choosing releases without recommendations 33

2.6 MRF: an example of Meta-recommendation message exchange 34

4.1 Avoiding recommendation loops 46

4.2 Example of metarecommendation message exchange 48

5.1 Adding reviewers: equal initial weights (all 100%) 65

5.2 Adding reviewers: random initial weights 66

5.3 Adding reviewers: equal vs. random weights 66

5.4 Removing a reviewer: comparing approaches 67

5.5 Deleting a reviewer 68

5.6 Number of operations performed by all peers (combined) per time 69

5.7 Reviewer usage by peers 70

5.9 NPMRA: release usage over time 71

5.8 Removing reviewers: Message-Aging approach 71

5.10 NPMRA: releases usage and over-all ratings 73

5.11 LLMRA casualties 74

5.12 Numbers of agents using reviewers 76

5.13 Reducing numbers of casualties 77

List of Tables

4.1 Global ratings 44

4.2 Trust weights for peer x 44

4.3 Calculated versions ratings for peer x 45

4.4 Trust weights for peer y 45

4.5 Calculated versions ratings for peer y 45

4.6 Updated reviewers trust weights for peer x 46

4.7 Adding a new reviewer 47

4.8 Trusted reviewer lists for agents A (left), B (middle), and C (right) 49

4.9 Trusted reviewer lists for agents A (left), B (middle), and C (right) 49

4.10 Trusted reviewer lists for agents A (left), B (middle), and C (right) 49

4.11 Agent view using LLMRA 53

Acknowledgements

Finishing this research study could not have been possible without the guidance of my advisor, Dr. Silaghi, who encouraged me in focusing on what matters most to achieve the desired final outcomes.

I am very grateful to all my teachers who helped and guided me throughout my studies, and to the future researchers who will build on this research and extend it.

Dedication

To my father, my brother Jamal, and my sister Amal who will always live with me.

To my family who supported me to conduct this research.

Chapter 1

Introduction

Free and Open-Source Software (*FOSS*) applications are considered a viable alternative by many users, with some applications providing levels of performance and security that rival – and even exceed, in some cases – those of their proprietary/commercial counterparts. Even when “free” refers to “freedom” rather than “gratis”, FOSS applications still tend to cost less, in general [64]. However, some challenges are more evident in this paradigm. For example, the decision whether to update an application to a newer version is not an easy task, and can have serious implications. The lack of a legally binding contract between the developer and the end-user allows the developer to drop certain functionalities or stop providing support for some features in their software (for example, because of changes in the developers’ vision) regardless of the end users’ requirements. In other cases, and especially in an unstructured and decentralized development model (no central maintainer), there can exist multiple competing releases (distributions or *distros*) of the same application (forks)

from different developers. While the alternative applications might be fundamentally different in many aspects from each other, the developers might distribute them under the same name, and/or with overlapping version numbers.

Security is always a major concern in software development. Attackers might provide their doctored versions of common FOSS applications that look and function similar to the original ones, but with hidden features that allow the attackers to access sensitive data, for example, or even take full control of the attacked machines. In a similar way, a developer of a commercial software might intentionally provide broken FOSS applications, to discourage users from continuing using them.

In mission-critical systems, or systems that handle users' sensitive data, it is a common expectation to provide secure software updates, preferably, in an automated way, to agents in order to resist any potential attacks. Failing to properly secure this process itself, for example in a petition drive system such as *DirectDemocracyP2P* [54, 55], attackers might deploy doctored releases of the system that allow them to disrupt a voting process.

Software updates are essential to improve performance, fix bugs and security vulnerabilities, or add new functionalities. Because of the open nature of the Free open-source software (FOSS) environment, the process of updating a software faces unique challenges, and can be susceptible to attacks that aim to influence the users to adopt certain versions of the software controlled by the attackers. In addition, multiple versions (or *distros*) from different developers can be found. This makes it difficult for users to choose the best version/distro for their use.

Recommender Systems (RSs) can provide an effective solution to these situations, however, extra efforts should be made to provide robustness to attacks and to cope with the variety of unique challenges in FOSS development.

In current literature, there is no widely accepted decentralized development model that guarantees complete ownership of the project by the community, and that truly preserves the *open*-ness aspect of the FOSS development process. Most FOSS applications utilize a centralized model, with one or more maintainers that control the integration of all patches into the main release.

An existing research that addresses the aforementioned issues is the *meta-recommendation* framework presented in [6, 7], which aims to automate the process of updating the P2P FOSS application to newer versions, while providing an adequate level of protection against the threats mentioned above. To make the decision of whether to update to a certain version or not, the system utilizes independent reviewers that test the different versions of the FOSS, and digitally sign and publish their findings. Those reviewers can be chosen by peers manually if they have, for example, a direct relationship to that reviewer, as being a co-worker, friended on a social network, a known organization, ...etc. They can also be contracted to provide their professional reviews.

The Metarecommendation system also allows peers to utilize recommendations from other connected peers (weighted relatively with their distance, measured using some chosen function) to automatically choose their trusted reviewers.

While this design can provide an improved robustness against some attacks, for example the *Stacking The Deck Attack* [6], it does not handle dynamic environments, namely, when new reviewers join the system, or when participating reviewers leave. The second case, in particular, poses a greater threat, where we noticed that peers can keep exchanging messages about reviewers that are no longer active. That means the removed reviewers still can influence peers' decisions, even after being removed.

In this work, we expand the *meta-recommendation framework* of [6, 7, 37] to mitigate the challenges due to churning of *reviewers*, mainly, the arrival and departure of reviewers in a peer-to-peer recommendation system. This framework provides a mechanism for agents (peers) to automatically make their decisions on whether or not to upgrade to a new release or distribution of a Free and Open-Source Software (FOSS) based on recommendations from reviewers. The reviewers themselves are also feedback from expert user agents, and these reviewer ratings (the *meta-recommendations*) are then exchanged by user agents. Each agent can build their database of trusted reviewers, and then uses the ratings from these reviewers to choose a distribution and a release of the software in question.

The above framework has been shown to provide resistance against attacks that try to influence agents to adopt compromised releases of FOSS, such as the *Stacking the deck attacks* described in [6]. However, these experiments were conducted assuming a static set of reviewers. In particular, reviewers who leave the system can have their previous reviews still being used by users who record them as trusted reviewers.

In this work, we study the effectiveness of this framework in a dynamic environment, where reviewers can join or leave the system at any arbitrary time, or can be blacklisted/discarded by agents, if they are shown to make poor recommendations. Poor recommendations can occur if the reviewers have been taken over by an attacker or if the reviewers are disseminating outdated quality information. We propose algorithms which handle adding new reviewers to the network and disseminating their new ratings, as well as removing outdated reviewer information.

In general, dynamic environments present some unique challenges: when a new user joins, a recommender system has no record of that user's preferences (a cold start situation [31] [12]), which makes it difficult (and less accurate) to recommend a particular version to that new user. The same is also true for new software releases that have not been reviewed/rated yet).

An ideal framework for decentralized P2P dynamic FOSS environment aims to:

- Strike a balance between system performance and accuracy of recommendations
- Handle dynamic environment changes, where users and reviewers may join or leave at any time, and handling the related challenges such as cold start situations
- Resist attacks aiming to influence users to install doctored updates that benefit the attacker

In this research, we use a set of key roles defined below, some of which are going to be discussed in more details later. Note that the following concepts are sometimes not consistently used in the FOSS culture:

- **Contributers:** participate in the development of FOSS, by providing new functionalities, or fixing existing bugs.

- **Reviewers:** test and rate software releases. Produce a signed test report (a review) that is going to be attached to the source and/or binaries of that release provided by the *distributors*.

- **Distributors:** provide the downloadable source and/or binaries of various releases, along with their respective reviews from reviewers. Distributers also act as download mirrors.

- **Users/Peers:** end users of software releases. Distributers also rate reviewers and exchange those ratings with other connected users (peers) in the P2P network.

1.1 Meta Recommenders

Recommenders (reviewers/testers) can be considered as items, so they can be rated based on their recommendations, and top-rated recommenders can then be recommended to other users by the *meta-recommenders*. This is shown to significantly reduce the impact of attacks based on controlling or replacing some reviewers in order to promote a doctored version of the software. This is also an effective solution for new users joining the system that have not established preferences yet (mitigates the cold start situation).

1.2 Privacy in Distributed Constraint Satisfaction Problems (DisCSPs)

DisCSPs are widely used to model many distributed problems. The distributed recommendations can be viewed as a DisCP where the recommendation has a newer version of the software that also satisfies all the required functionalities set by the user, and has a higher rating than other competing versions.

The framework we proposed in [43, 43, 46] to cope with privacy in DisCSPs and the modified common algorithms used to solve these problems to utilize the mentioned framework (including SyncBT, and ABTU) are shown to significantly minimize privacy loss of agents.

We introduce Utilitarian Distributed Constraint Satisfaction Problems (UDisCSP), an extension of the DisCSP that exploits the rewards for finding a solution and the costs for losing privacy as messages are exchanged between agents.

1.3 Contribution Summary

- Detection of Churning Problems One of the main goals of this study is to identify the issues related to churning of reviewers. Previous work did not consider dynamic environments with reviewers joining or leaving the system. Special attention is paid

to the case where reviewers leave the system (or get removed) but their recommendations remain in circulation by peers who are not directly connected to them, and therefore, keep these phantom reviewers in use by the system.

- Framework for modeling trust with churning Adjustments to the metarecommendation framework are proposed and tested to ensure the aforementioned framework can coop with churning of reviewers. We make sure reviewers no longer in the system are purged as soon as possible to avoid issues described under the previous point.

- Solutions and algorithms We propose algorithms to accommodate churning and keep the systems relatively convergent even with the presence of churning. We cover a few different types of algorithms, ranging from basic ones that can be easy to implement and t integrate into existing frameworks, to a more efficient algorithms that can be more scalable to accommodate scale-free systems.

- Evaluations (experiments) We propose and report the result of different experiments to test the proposed solutions.

1.4 Motivation

Software Updates can contain bugs and disrupt users, as in the recent case of Android 10 update [1], CamScanner [2], or Google Chrome White Screen Of Death [3]. Systems such as DDP2P can be taken by political opponents and groups of interests.

To develop an attack-resistant framework for FOSS updates, we propose a *Distributed Recommender System*. The proposed distributed RS for software updates is suitable to be used in a P2P FOSS environment, and can be more robust than traditional centralized RSs. Recommendations in the proposed system are not made by a limited number of reviewers/testers (since they are dynamic).

1.5 Future Work

We intend, in future work, to investigate in more details the proposed approaches, and compare them in terms of efficiency and robustness in environments with different types of social networks. We also intend to utilize a distributed ledger technology such as *blockchain* to share agents' data to improve efficiency by minimizing the number of exchanged messages.

We also consider possible applications in embedded systems networks or IOT devices as some of the proposed algorithms shown to require minimal network and processing overhead. We will report such results a in a future work.

1.6 Chapters of the Dissertation

After introducing the related literature and background concepts related to software updates in FOSS, the Stacking the Deck Attack, and meta-recommender systems, we describe the concepts at the center of this study. We cover in details the algorithms proposed in this

study. We then report the theoretical and experimental results concerning churning of reviewers. , also presenting conclusions and future work.

Chapter 2

Background and Literature Review

2.1 Free and Open-Source Software (FOSS)

Free and Open Source Software (FOSS) is becoming widely used by individuals and developers, in enterprises and start-ups [14]. Studies like [18] show that this trend holds in the US and in the European Union. The FOSS development model allows users the freedom to modify, enhance, and redistribute the source-code [25, 48]. Some of the advantages of adopting this model include free or low reproduction costs [21], faster software bug fixes, and the ability to share implementations from the community and/or academia [27, 51].

Free and Open-Source Software (FOSS) utilizes a variety of collaboration models [62, 29], most commonly, the centralized one, where the project's *Maintainer* (usually the developer(s) who started the project) controls the main repository and needs to approve any patches (from the community or the co-developers) in order to be included in the main release. Some projects have multiple maintainers, and involve other trusted developers in

the patches review process. Others employ a *peer-review* approach. Figure 2.1 shows some of the basic tasks and roles involved in FOSS development and updates.

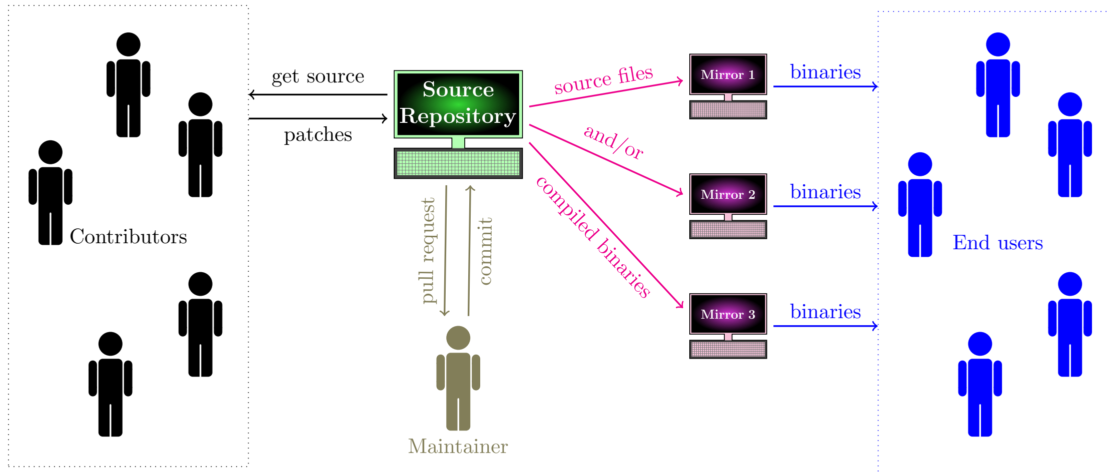


Figure 2.1: Centralized FOSS development model

The main maintainer needs to approve all contributions (patches) from the community before up-streaming them.

Free Open-Source Software (FOSS) represents a significant movement in software development [49, 28]. The most important aspect arguably is the availability of the source code to be accessed and freely modified and redistributed by its developers [26, 22] under various licenses, such as the different versions of GNU General Public License (GPL). The many success stories of FOSS have proven that high quality software can be produced by adopting this paradigm [35], and they can form a viable alternative to commercial products. Examples of such success include the *Apache web server* software, *Firefox* web browser, *Open/Libre Office* suites, and the *Linux* operating system kernel. Some of these projects

contain millions of lines of code contributed by thousands of developers (mostly volunteers) from all around the world. These mostly unpaid volunteers contribute in developing FOSS applications for a variety of motivations [63, 50, 10].

However, some aspects of this paradigm are regarded as disadvantages by many of its users [9]. Most noticeably, the lack of a legally binding contract between the developers and the end-users, which allows for dropping features that are considered important by some users, or the addition of experimental code that might not function as the users expected. Another common change in FOSS projects is the change of vision for the project as a result of being taken over by different developers with different goals.

In some cases, the developers can, whether intentionally or not, introduce security vulnerabilities [25, 15]. While new releases are effective opportunities for the developers to fix existing issues, the very nature of FOSS also allows for developers to embed vulnerabilities [8, 23]. For example, Android 10 update [1], CamScanner [2], or Google Chrome White Screen Of Death [3]

We focus in this study on the decentralized development model that eliminates the role of a central *maintainer*, which can be one of the most vulnerable points in the system. That, however, presents some unique challenges [32]. An example is when multiple developers work on the same project, without coordinating with each other, and they might produce different releases with different features, but sometimes have conflicting version numbers. That, in turn, makes it difficult for the end-users to choose the best and most current version

that they should use. Some attackers can introduce malicious applications as newer versions of popular FOSS applications, which represents a major risk to users' security [39].

2.2 Updates in Free and Open-Source Software

In addition to fixing bugs and improving performance, new software releases can also introduce new issues, remove features, or change the behavior of the software [8, 23]. This is particularly important in FOSS projects, as the over-all quality of the project is shown to be related to updating practices [34, 33]

Software updates not only offer the newest features and improved performance, but they also offer important bug fixes and deal with serious security vulnerabilities [17]. Developers usually encourage users to update their software to minimize the number of the different co-existing versions for the same software, which can have a negative effect on interoperability, and complicates software maintenance.

These updates are usually distributed as patches that clients can download and install (other types of media were more common in the past, or when there is no Internet connection available) [42]. Most software developers prefer to automate update distribution (auto update) by a mechanism where a client software (namely, the *update manager*) checks periodically for new updates (or gets notified whenever there is one), and then downloads and installs that update, after being properly authenticated.

In proprietary software paradigm (closed-source software), the source code is not publicly available (examples include Apple Mac-OS, Microsoft Windows). Various software

producers use different methods [8] to authenticate and deploy updates. The solutions commonly distribute binaries. A *public key* is frequently used to sign the update, and the transfer may or may not require an encrypted connection. The packages to be downloaded and installed are usually determined by running an update client on the user's side, update servers (mirrors) are then contacted to get the required packages.

In the FOSS model, developers offer updates in a variety of ways ranging from manually downloading and installing patches, to an entirely automated updating process. An example of the latter is used by *Debian Linux*, where a package manager is used to determine which packages need to be updated by checking the repositories that can be configured by the user.

In the common FOSS paradigm, developers collaborate in developing the software. The main repository is kept by the *Trusted Developers*, known also as *Maintainers* (usually, the ones who created the project), and only they can approve and integrate newer or improved features from other developers (also referred to as *Contributers*). The *Distributors* can then access the main repository to download the latest source files, and compile/build the distributable binaries, and post them on *Mirrors* for users to download and install.

Figure 2.1 shows how FOSS development and updates work in most environments, where a centralized *Maintainer* needs to view and approve all patches from other contributers (co-developers from the community) before they become part of the main release.

2.3 Attacks in Software Updates

In this research, we focus on the attacks related to decentralized P2P environments, in particular, we focus on the *Stacking The Deck Attack* described in [6]. In this attack model, developers are targeted or replaced with others who work for the attacker.

In a *centralized* environment [61] (with one developer/provider and many users/clients) the most common form of attacks is when the attacker impersonates the legitimate developer's update servers (*man-in-the-middle* attack), by intercepting and modifying messages between that server and the clients or by introducing a fake update server via DNS cache poisoning [16]. The attacker -in this case- can perform any of the following attacks (as described in [16]) on the software update process or against the update systems:

- **Arbitrary package** The attacker provides a package they created
- **Replay Attack** The attacker provides an older version of the package
- **Freeze Attack** The attacker prevents the client from seeing new packages
- **Extraneous Dependencies** The attacker adds its own additional packages as dependencies
- **Endless Data** Attacker provides an endless stream of data to crash the client's system

On the other hand, in a *decentralized* environment, more commonly adopted by the FOSS development paradigm, users rely on reviewers who test and rate the different releases from different developers, and recommend to users the versions best suited to their

needs. Those reviewers are vulnerable to a variety of attacks as well, mostly aiming to influence users to update their software to a doctored version controlled by the attacker. An example of the aforementioned attacks is *Stacking The Deck Attack*.

In addition to getting the improved performance and added functionalities that are usually offered by the newer releases, most users update their software to remedy security vulnerabilities, and potential privacy threats. However, the update process is not without risks, whether in open source, or closed source (proprietary) software models [8, 60, 36].

[16] presents a study of some popular package managers (both open and closed source), and their vulnerability to some attacks like the *man-in-the-middle* attacks.

Stacking the Deck Attacks *Stacking The Deck Attack* described in [6] is an example of attacks that threaten decentralized P2P FOSS environments. In this attack model, developers and reviewers are targeted to be replaced with others who work for the attacker. The newly inserted reviewers will then try to influence the end-users to update to a new (and doctored) version of the application developed by the attacker.

Example 1 *As an example of a decentralized FOSS application, we consider Direct-DemocracyP2P (DDP2P) presented in [55, 54], which is a P2P system used primarily for petition drives. the following scenario helps demonstrate how this attack works.*

- *Users might want to use the DDP2P system to get signatures for a petition that might conflict with the interests of an influential organization, say Snake Oil, Inc. One possible way for Snake Oil, Inc to disrupt the petition drive process is to attack the DDP2P software using its automatic updates.*

- *Snake Oil, Inc could disrupt the petition drive process by attacking the DDP2P software using its automatic updates.*
- *By replacing the project's maintainers with people they control, they can introduce a newer version with features that disable or disrupt the targeted petition. That can be achieved by offering the original maintainers (the volunteers) a good paying job in another project, for example.*
- *Once Snake Oil, Inc is in control of the DDP2P development process, it can introduce a newer (updated) version with features that disable or disrupt the targeted petition drive.*
- *In a traditional environment, most users will automatically update to the newer doctored version, and that might result in altering the vote counts or changing the voting results in a significant way that benefits that organization. The effect could also be more subtle and less noticeable such as slowing down the communication of votes for the targeted petition, for example.*
- *Without an agreed upon mechanism to warn other users, this attack can take a long time for most users to detect it (especially if the effect is subtle).*

This situation can be mitigated by inserting independent *Reviewers* between developers and end-users. By experimenting with the various releases of the FOSS, some of these reviewers can detect such changes in the software, and rate it accordingly. If an attack is

detected, other volunteers can then start their own development branches based, for example, on the last known good (not doctored) version of the software, which can be achieved relatively easy since all the needed software is open-source.

An essential assumption here is the independence of the aforementioned *Reviewers*, which forms the base for the trust in them by the users. Users can introduce *Reviewers* they know personally, or have legal contracts with, as trusted reviewers. They can then recommend those reviewers to their connected peers. If these reviewers get adopted by peers, those peers can recommend them -in their turn- to their own connected peers, and so on. An *Amortization Factor* (for example, 0.95) can be used to reduce the trust weight with each step. This trust-based model was shown to provide resilience of the system in front of various attacks. Some variations of this model along with their overall effect on the stability and performance of the system are discussed in [6] .

2.4 Recommender Systems

Recommender systems are widely used to introduce/recommend interesting items (products, movies, services, etc.) to individual users based on their taste [40], i.e. their ratings of previous items. They are useful also to sellers, as targeting the consumers that are most likely to buy those products is expected to increase their profits. Based on how recommender systems make their recommendations, one can classify them [4, 41] as:

- **Content-based.** Items recommended based on their similarity to the ones the consumer liked, as they hinted by these consumers giving them high ratings.

- **Collaborative Filtering (CF).** Items recommended based on their ratings by other similar consumers, who have similar taste.

- **Hybrid.** Both previous methods are used simultaneously.

Some of the attacks (like Shilling attacks) that might target recommender systems are aiming to influence the recommender system [5] to maximize the ratings of certain items (e.g., by submitting fake positive ratings for those items), which is known as *push* attacks. Other reviewers might want to decrease the ratings of other products, referred to as *nuke* attacks. Other attacks might target the recommender system itself, to affect its performance and/or accuracy, making users loose trust in its recommendations.

Recommender Systems (RSs) are used widely to help users/consumers find products and services [13], or choose from available ones without the need of extensive personal experience [41]. They have been extensively studied for a variety of applications ranging from on-line stores, to search engines.

In its purest form Recommender Systems are clever updates to the traditional “Top 10 list” in that these systems tailor the recommendation to the particular user using data that the system collected. Consequently, Recommender Systems (RS) are useful software tools which provide users with suggestions for items that may be of interest. Often these suggestions lie in the realm of decision making (e.g what product to buy, what news channels to subscribe to or what music to play next). Each Recommender System itself is specifically divided by product type (e.g news channels recommender system will suggest news outlets based on the user’s metrics). The preponderance of usefulness of RSs lies in the

fact that the RS can be integral in decreasing the overwhelming amount of information a user may encounter. This allows the user to make choices regarding items that are presented [41]. A prime example of software using a RS is the e-commerce site Amazon.com. Amazon.com filters previous purchases of users and personalizes the online store for the particular customer.

Recommender Systems can be divided into two broad groups [52, 11]; systems that are "Personalized" and systems that are "Non-Personalized". Non-Personalized systems do not take the particular user's preferences into consideration. Instead, such systems simply recommend something based on large quantities of overall positive reviews that are readily available. With this type of system, if the majority enjoyed this item, the item will appear in the list of recommendations. With Personalized systems each user becomes the center of consideration. The idea here is to bring the "right" item to the "right" user at the "right" time [24].

RSs can be targeted by a variety of attacks (as discussed in [38, 30]) that attempt to influence the recommendation of certain items, either by increasing their ratings (*push* attacks), or decreasing them (*nuke* attacks).

2.4.1 Recommender Systems Uses

Recommender Systems (RS) are useful mechanisms which provide users with suggestions for items that may be of interest. Such systems are classified based on the information

used as a part of the recommendation step [24]. *Content-based filtering* will gather details regarding the “content” of the data such as the title or cost of an item. When this particular information is processed the result provides useful features/elements from that content. *Collaborative filtering* instead bases its recommendation on other similar users. One particular application will determine relationships between users (e.g compute a particular user’s correlation with other users in the system and suggest items purchased by the related users). Another application computes correlations between the items in the system to suggest novel items to a user [24].

While the framework discussed here cannot be considered a conventional RS, it can be regarded as a distributed heuristic-based hybrid meta-recommender system. On one level, rating the reviewers is based on other users’ ratings (CF recommendations), and on the next level, the reviewers rate the various software versions based on their properties and features (Content-Based recommendations).

2.4.2 Recommender Systems Types

Most studies (as in [4]) classify them into the following categories: Content-Based (which makes recommendations for new items based on the similarity between those items and the ones known to be given high ratings by the consumer), and Collaborative Filtering (CF), which does not require the system to have detailed information about the items, but instead uses the ratings given by other consumers, who previously exhibited a similar taste. Hybrid systems utilize both methods.

Most recommender systems utilize a client/server paradigm, where one centralized server collects information from each client, and produces recommendations based on that collected information. The main disadvantage in this paradigm is the vulnerability to attacks that might threaten the data of the users of the system, and attacks aiming to disrupt the system. For example, a Denial-of-Service (DoS) attack on the single server could affect the entire system. Meta-recommenders can support decentralized systems with distributed recommendations from independent reviewers, which is most suitable to a peer-to-peer (P2P) paradigm, such as the one used by *DirectDemocracyP2P* and the Blockchain [37]. A different usage of the term Meta-recommender system is used with the meaning of aggregating reviews from multiple sources for customizing recommendations [53].

Furthermore, RS can be classified based on the information used as a part of the recommendation step [24]. Content-based filtering will gather information regarding the "content" of the information such as the title or the cost of this item. When this particular data is processed the result provides useful features/elements from that content. Collaborative filtering instead bases the recommendation on other similar users. One particular application will determine relationships between users (e.g compute a particular user's correlation with other users in the system and suggest items purchased by the correlated user). Another approach computes correlations between the items in the system to suggest novel items to a user. Similarly, hybrids of these systems have also gained popularity, specifically a mix of Content-based and Collaborative systems [24].

Each RS type has its own specific advantages and disadvantages. For example, though the Content-Based approach is independent of other users and can help pick items within the user's distinctive preference, it is limited by the features defined within the system. As its complement, Collaborative filtering does not need the features of the items and depends largely on the community of users. Conversely, with Collaborative filtering novel input cannot be recommended due to the fact no correlation will exist with similar items [24]. Nonetheless, the issues with RS do not simply lie in the set-up and data analysis but also within security. In particular, collaborative filtering has recently been shown to be vulnerable to a "shilling" or "profile injection" attacks. In these attacks a malicious user makes a biased profile in order to elicit a particular behavior. Since collaborative filtering collects a plethora of user profiles for analysis, it is possible that within this database of profiles are biased profiles which may be inevitable considered by genuine users, resulting in an overall biased choice. In effect, this renders the whole "trust" of a recommender system questionable. Previous research into shilling attacks looked into the four common types of attacks (the random, average, bandwagon and segment attacks) as well as different algorithms for detection, such as the KNN and Singular Value Decomposition [65].

2.5 Meta-Recommendations Framework (MRF) for FOSS Updates

The meta-recommender framework introduced in [7, 37] was shown to provide a mechanism to minimize the effects of certain attacks on the reviewers. That was achieved by treating the recommenders (reviewers) themselves as items, so they can be recommended to other users. These *meta-recommendations* are based on peer adoption of those reviewers. The aforementioned framework, however, suffers from some limitations in dynamic environments as described in more details in the next section.

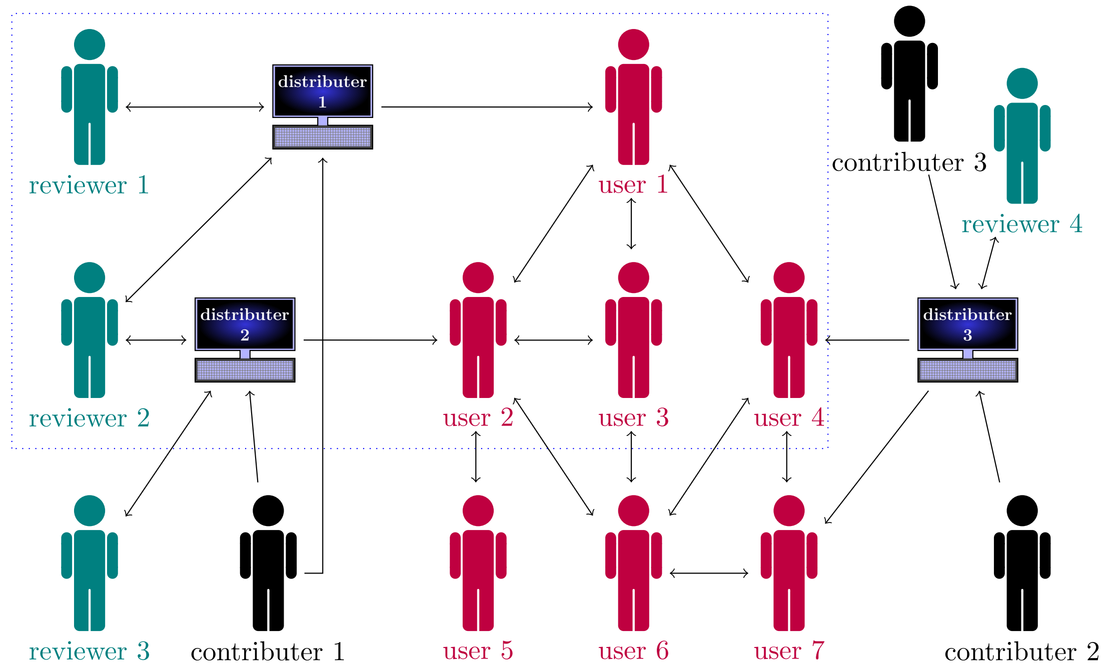


Figure 2.2: Decentralized FOSS updates framework

Reviewers test various releases and report and sign their findings.

2.5.1 Trust Propagation in the Meta-Recommendations Framework

Peers exchange messages to communicate information about their trusted reviewers. They also exchange data about the distributors that can be contacted to view the test results , and to download the source or compiled (binaries) files for the various releases. The reviewers themselves are being treated as items, so they can be recommended to other users. This recommendation process is explained in more detail in Figures 2.2 and 2.3.

Figure 2.2 shows the framework previously proposed for decentralized FOSS updates. This framework supports multiple decentralized *Maintainers* developing their own releases. *Contributers* can participate in releases, as before. Releases are tested by independent *Reviewers*, and the results of those reviews are posted on the distribution mirrors (*Distributors*), along with the relevant binaries for those releases.

Figure 2.3 gives a closer look to a specific part of the previous figure (the top left dotted rectangle from Figure 2.2), which reveals some of the messages exchanged between users (peers) that include their discovered *Distributors*, and trusted *Reviewers* (with their respective trust factors or weights).

Each peer maintains a list (usually of limited length) of trusted reviewers. Only the test results from those reviewers are taken in account when deciding to update to a certain version. That means reviewers with higher trust weights (for example, those introduced by directly connected peers) are going to have bigger influence on the decision than reviewers with lower weights (for example, those introduced by remote peers, in terms of the used

distance function). This list is updated as new messages arrive from all connected peers, and is communicated to the other connected peers.

In the original static scenario, at time 0, some peers can manually set their initial trusted reviewers (from a prior social connection, or a contracted tester). As time passes, peers communicate their trusted reviewers to each other, along with their respective trust factors (which are amortized by a predetermined factor with each step, 0.95 in this example). Each reviewer is considered only once, in case it was recommended from multiple peers, we take the one with the highest trust weight, as shown in the case of *user 3*, which gets messages about *Reviewer 1* both from *user 1* (with 76%), and from *user 2* (with 71%), so it takes *user 1*'s recommendation as it has the highest trust weight.

Peers communicate only the N most trusted reviewers (those with the highest trust weights). This reduces the number of exchanged messages between them. A similar approach is used with distributors. The meta-recommendation function used by prior work is:

$$\mathcal{W}_{p,\tau_i,\Omega} = f \cdot \left(1 - \frac{\underset{j\in\Omega}{\#}\{j \mid \tau_i = \tau_j\}}{K \cdot n}\right) \underset{j\in\Omega}{max}\{\mathcal{W}_j \mid \tau_i = \tau_j\}$$

where $\mathcal{W}_{p,\tau_i,\Omega}$ is the weight of the recommendation made to peer p for reviewer identity τ_i based on the set of n reviewers described in knowledge base Ω, f is an amortization factor enforcing proximity, # is the cardinality function, $\mathcal{W}_j$ is the weight peers assign to j, and K is a diversity factor.

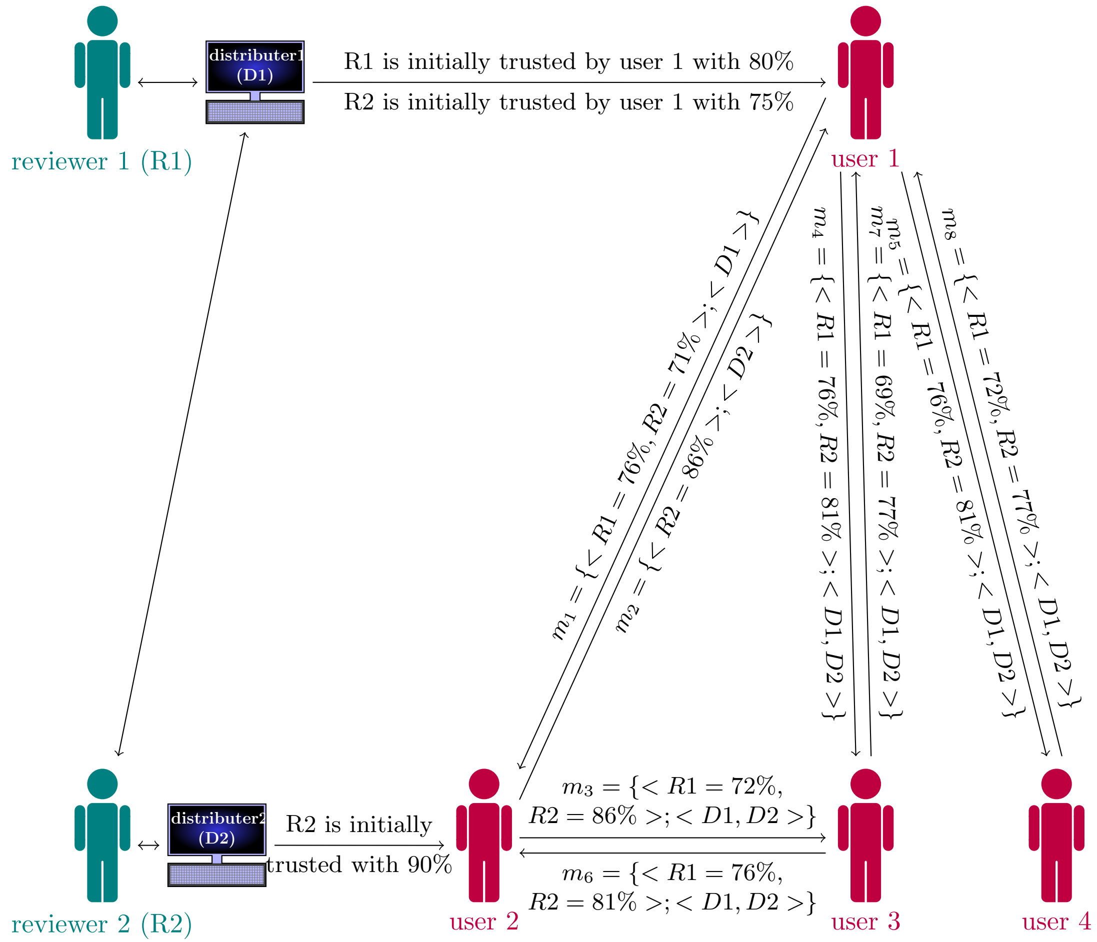


Figure 2.3: Messages exchanged between peers
Messages are used to communicate discovered distributers and trusted reviewers (using amortization factor of 0.95).

2.5.2 Meta-Recommendations Framework (MRF) Concepts

For the mechanisms described in [6, 7, 19], each agent in the system keeps a list of known reviewers, along with their respective trust weights. Reviewers are initially introduced to the system with a predetermined trust weight by agents who know them personally, or have other means of ensuring their trustworthiness. Agents periodically exchange their lists of

trusted reviewers. With each hop, the trust weight of a reviewer gets reduced (multiplied by an *amortization factor* λ = (0..1]). Agents sort all the reviewers they know about in descending order of their respective trust weights, and keep only a predetermined number N of the highest rated reviewers. The *Meta-Recommendations* are a list of tuples in the form $\langle reviewer_i, rating_i \rangle$, where $rating_i$ is the agent's rating for $reviewer_i$. *Attackers* are reviewers who try to persuade agents to switch to a compromised release of the software (i.e., to become *casualties*). The meta-recommendations framework aims to mitigate attack effects by minimizing the numbers of casualties.

Figure 2.4 illustrates a comparison of the number of casualties in case of an attack when using MRF and a base line algorithm (BLA) that picks randomly one of the available distributions and releases. In this example, 10% of the reviewers are considered attackers (starting at $epoch_{20}$). While in both algorithms there is an increase of casualties, MRF reduces the number of casualties by more than 90%.

Figure 2.5 shows the percentages of agents who ended up installing each release of FOSS when using BLA. The data suggests that each release's usage is proportional to how many times it was advertised. Figure 2.6 illustrates an example of *meta-recommendation* messages sent by agents to their connected peers to inform them about the reviewers they know about.

In the current model, these messages are exchanged periodically. Each peer handles messages received at t_i, and makes the required adjustments to its list of trusted reviewers

accordingly, then sends out its new updated list to its connected peers, to be processed at time t_{i+1}.

As we can see, the reviewers in this model are being treated as items, and can -themselves- be recommended to other peers with various trust weights. These *Meta-Recommendations*, in addition to the reviewer independence, provide an additional layer of protection that can prove very useful against the previously mentioned risks.

The following are some of the points we considered in this process:

- To keep the size of the messages manageable, peers send only the most trusted N reviewers (sorted by their respective trust weight)

- Messages about a trusted reviewer are not sent back to the source peer (the one who announced this reviewer). In the previous example, $user_2$ sends to $user_1$ announcements about $reviewr_2$ and $reviewr_3$, but not $reviewr_1$.

- If the same reviewer is announced by more than one connected peer, only the tuple with the highest trust weight is kept (we apply the same process for any future announcements regarding that reviewer). In the example in Figure 2.6, that can be seen in the case of $user_3$, which receives announcements about $reviewr_1$ both from $user_2$ (with trust weight of 86%) and from $user_5$ (with trust weight of 95%), so it keeps the tuple from $user_2$ and discards the other one.

- When the source peer changes its trust weight for a particular reviewer whether by increasing or decreasing it (the new tuple contains a previously announced reviewer ID, with a different weight), the receiver peer updates its list accordingly.

- The use of the amortization factor helps avoid infinite prorogation of messages in the system. It can also provide a slight advantage for local reviewers to be chosen (used) over distant ones.

- Messages can be sent at any time, but they will be queued by the receiving peer, and will only be processed in the next round. While the algorithm is synchronous, this allow peers to communicate even if they are not perfectly synchronized.

The recommendations been exchanged conform to a restricted language, weaker than first order logic, and therefore they cannot lead to circular reasoning.

This meta-recommendations mechanism provides a way for peers to share their experiences and ratings of the reviewers. For example, if a reviewer starts to show a bias towards a certain software release (say, because of an expected financial gain), the peers can lower their trust weight for that reviewer, and communicate that to connected peers, who forward the change to their connected peers, and so on. In similar way, this can provide a significant resistance to reviewer take-over attacks as discussed in [6, 7].

2.5.3 The Meta-Recommendations Framework Roles

The Metarecommendation framework (MRF) [6, 7] introduces the following roles:

- Independent *Reviewers* (shown in Figure 2.2), will be discussed in more details in the following chapter.

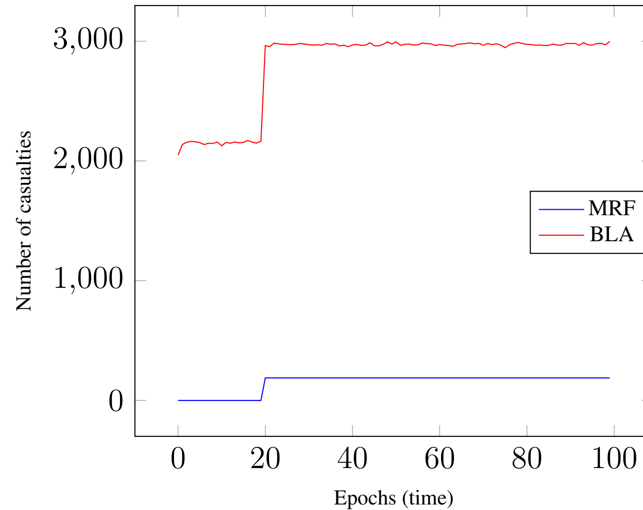


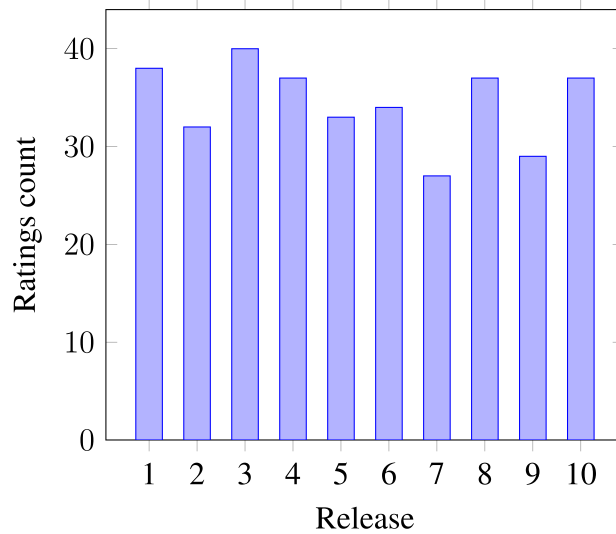
Figure 2.4: BLA vs. MRF
numbers of casualties when 10% of reviewers become controlled by attackers at $epoch_{20}$

- *Distributors* they can be compared to download mirrors. Their main task is to provide the source files, the signed binaries, and the test results reports signed by the independent reviewers.

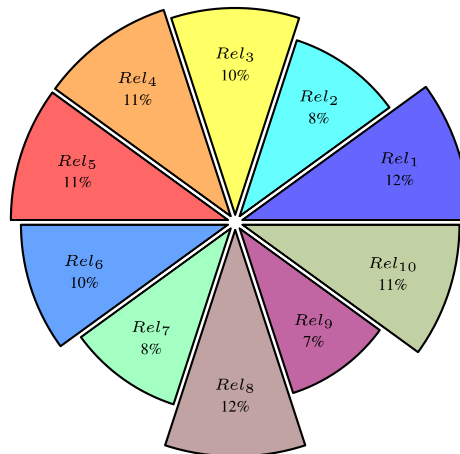
- *Peers* exchange messages to communicate their trusted reviewers to other peers (the *meta-recommendations*). Each message consists of a set of tuples, of the form <reviewer ID, trust weight>. Figure 2.6 shows some of these messages (chosen from various time steps). In this example, we use an amortization factor of 0.95.

2.5.4 Privacy in Multi-Agent Systems

Recommendations made in a dynamic environment can suffer from loss in privacy of users [59]. For example, some attacks, such as the *new group attack* described in [20],



(a) Releases initial ratings counts (at $epoch_0$): showing how many times each release was advertised



(b) Percentages of agents ended up using each release (BLA)

Figure 2.5: BLA: choosing releases without recommendations

can exploit the dynamic nature of the environment to deduce the preferences of individual users by comparing their different contributions in dynamic recommendations. This cannot be done as easily in a static environment, where the same users contribute to all recommendations. Knowledge representations schemes, e.g. *constraints*, can be used to specify desired properties expected by various players from the recommendation system.

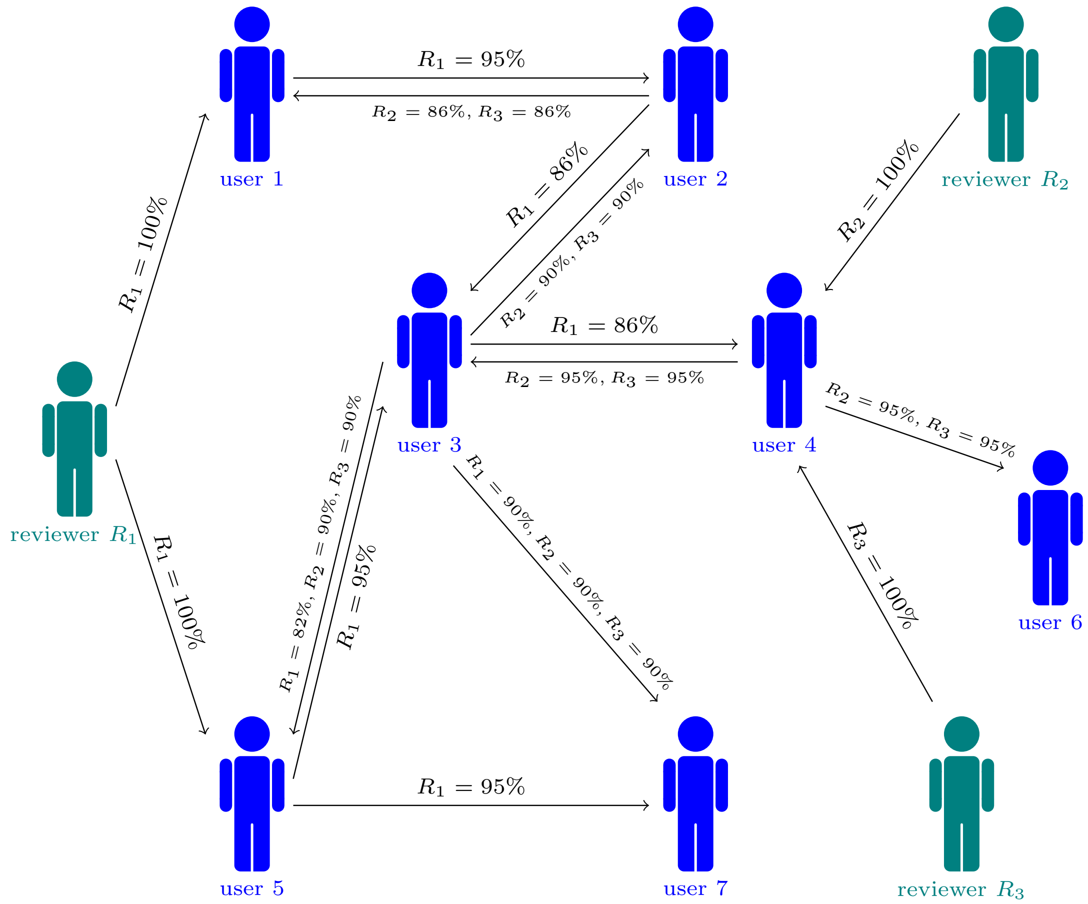


Figure 2.6: MRF: an example of Meta-recommendation message exchange
peers exchange messages with reviewers IDs, and their trust weight (using an amortization factor λ of 0.95)

In distributed CSPs, openness in asynchronous search is investigated in [57] with focus on agents joining and leaving an ongoing search process without affecting the consistency of other agents.

It was also shown that privacy of distributed problems (modeled by Constraint Satisfaction Problems [56]) can be improved by assigning utility values to the various solving steps [44, 47] or by applying stochastic game methods [46, 58, 45]. Similar results can also be achieved by applying planning methods to the solving process [43].

Chapter 3

Concepts

In this work, we focus on dynamic changes in reviewers in the *Meta-Recommendations* framework. We consider in particular the following situations:

- New reviewers join the system, or regular users become reviewers.
- Reviewers (possibly currently in use by some peers) leave the system. This could be because the reviewers no longer want to participate in the process, or as a result of network connectivity issues.

We handle new reviewers in the same way we handle existing ones: they have to be introduced manually by a peer with a trust weight high enough (usually, 100%) that makes other peers use and forward them. If the initial weight is low, the introduced reviewer might be ignored by the receiving peers (if they already have N trusted reviewers with weights greater than the new one). Associating a *source* with each entry allows peers to keep track of the up-to-date weight as set by that source.

The latter type of changes (the removal of reviewers) presents the highest level of threat, since peers that continue to use an attacked reviewer, might adopt a doctored version of the software, and become a victim (or *casualty*) of the attack. To handle that, we consider the following approaches:

- **Self-announcement**
- **Drop/remove messages**
- **Limited-Life Meta-Recommendations**
- **Non-positive Meta-Recommendations**
- **Update-driven Meta-Recommendations**

3.1 Effects Due to Reviewer Churning

Deploying the MRF framework in a dynamic P2P environment can face some serious challenges. Two main challenging situations identified here are:

- Reviewers leaving the system.
- New reviewers joining the system.

We notice that in the completely decentralized environment proposed in [6], peers can keep exchanging messages between them about reviewers that are no longer accessible and active.

In this work, we evaluate the behavior of the previously studied systems for the latter mentioned kind of issues: namely caused by reviewers leaving the system. We report experiments confirming the extent of the problems that appear. To deal with these situations, we identify and investigate some families of mechanisms to detect reviewers that are leaving:

3.2 Handling Reviewer Churning

In this work, we study the following approaches to handle churning of reviewers.

- **Direct self announcement:** reviewers willing to leave broadcast a global message informing all peers about their intentions. This approach can fail in case of unexpected disconnection from the network, for example. Moreover, attacked peers (or peers who work with the attacker) might block this type of messages, preventing them from being received by the peers.

  We note that one cannot use methods based on *self-announcements* by leaving reviewers for the case where reviewers can be corrupted by attackers, or disabled by other external causes (death, lost keys, job change). Some of the possible attacks against messages generated for such mechanisms can be thwarted by cryptography, and suffer from all common challenges to key management.

- **Mesh-based peer detection and announcement:** peers use messages to disseminate announcements about changes.

Methods based on detection and announcement by peer, suffer from challenges related to the difficulty of detection and of differentiating benevolent peers from attacker peers. A possible attack consist of claiming that uncorrupted reviewers have left while they are active. A potential fix is to create a mechanism that lets reviewers be informed on their supposed demise thereby enabling them to confirm their continuous presence.

- **Limited-Life Meta-Recommendations:**

  The third family of solutions is based on letting end-users detect themselves the prolonged absence of reviewers by finer resolution tracking of the activity of these reviewers and computation of delays. if peers do not hear from a particular reviewer for a long time (longer than the specified *timeout*). Information about such delays can be exchanged between peers as in the previous case. Messages can specify whether nothing new was heard from these reviewers for long periods of time, combined with timeouts.

  To prevent messages from circling the system indefinitely, the source appends a *time-stamp* to the message, and the receivers compare that time-stamp against a predetermined time interval (the *timeout*). Messages that exceed the allowed age are dropped automatically from the system. This approach is implemented in Algorithm 2.

- **Drop/remove messages:** While the original framework uses only one type of messages (to announce/add a new reviewer), we can introduce a second type of messages that tell peers to drop (stop using and forwarding) a certain reviewer.

This is an effective solution (as shown in Experiment 5.5). However, since the newly introduced type of messages behaves in a different way (it is not forwarded based on *fitness*), it requires different measures to control the propagation of theses messages, which can also have significant impact on system complexity.

- **Non-positive Meta-Recommendations:** Source peers can announce a change of the trust weight for the relevant reviewer, and set it to zero (or even a negative value), which will overwrite the previously announced weight, and most likely move it to the bottom of the trusted reviewers list. That makes the mentioned reviewer less effective in the decision making process, or might force it to be replaced by other more trusted reviewers.

  Simulation results (omitted for lack of time) show that this approach is the least effective one, because of the optimization measure used by the current framework, mainly, forcing peers to keep a limited number of reviewers (only those with the highest weights). Reviewers with zero or negative trust weights move to the bottom of the list, therefore, they get dropped relatively early, and are not forwarded to other peers. As a result, most peers will continue using them, because they are did not receive the notification about the change.

- **Update-driven Meta-Recommendations:** Agents send only what has changed since last round (last update sent out). This not only helps reduce conflicts of recommendations when multiple peers try to propagate the same change, but it also improves the efficiency by reducing the numbers of circulating update messages.

3.3 Concepts

We start by defining formally the involved software update meta-recommender concepts of user, reviewer, release, neighborhood, casualty, and the attacker model.

In this work's context, an *agent* is a peer user, potentially embedded into a software whose updates are being evaluated. The peers are connected in a graph G with vertices the set of peers P and arcs given by the communication channels C, forming an overlay network. The agents (also referred to as *users* or *peers* in this study) are formalized as the set $P = \{P_{i,j}\}_{i \in \{1..n\}, j \in \{1,p_i\}}$, where $P_{i,j}$ is the set of p_i peers in $neighborhood_i$.

A *neighborhood* is a cluster of peers where each peer has a higher probability of being directly connected to other peers in the same cluster than to peers external to the cluster. The set of neighborhoods is denoted $N = \{N_1, ..., N_n\}$.

A set of software *releases* $R = \{R_1, ..., R_r\}$, from various competing distributions, is being considered by peers for potential updates. Each peer uses a current release R_i. A "less recent" partial order is defined on these releases, and each peer using release R_i only considers automatic updates to releases R_j where $R_j \not< R_i$. Some releases are controlled by the attacker at time t, as specified by a function $A_t(R_i) : R \rightarrow Boolean$.

A *reviewer* is an expert agent E_i who can associate releases R_j with a quality valuation $T_{i,j}$. A quality valuation is an element of the vector space R^k, subject to a partial order based on Pareto optimality. The set of reviewers at time t is $E^t = \{E_{t(1)}, ..., E_{t(e_t)}\}$ where $t(k) \in \mathbb{N}$ is a function showing the k-th active reviewer at time t.

An *attacker-controlled reviewer* is a reviewer whose quality function is manipulated in order to subvert the behavior of the system, under the control of an attacker. A function $A_t(E_i) : E^t \rightarrow Boolean$ is used to specify whether E_i is controlled by an attacker at time t.

A *casualty* is a peer $P_{i,j}$ whose system is under the control of the attacker. This happens if the peer is currently using an attacker-controlled release. A casualty peer may spontaneously escape the attacker by migrating to a safe release based on a probabilistic function $S_e(P_{i,j}, \Delta t)$ based on the time Δt since contamination.

The attacker model adopted in our study is based on the following assumptions. The attacker is active and manipulating release quality estimations. The attacker communicates along the graph G, and respects the protocol.

Associating a *source* with each reviewer rating allows peers to keep track of the up-to-date weight as set by that source, along with the required modifications to the algorithm (shown, for example, in line 11 of Algorithm 1).

Chapter 4

Algorithms

One of the main drawbacks of MRF is the design for static reviewers. In this thesis, we investigate mechanisms that address dynamism in reviewer availability.

Next we introduce the ideas utilized by the new proposed algorithms.

4.1 Choosing a Software Release

Each agent chooses a release that maximizes the quality function Q in Equation 4.1. This function was designed to take into account both the reviewer's trust weight and its rating for a specific release.

$$Q(j) = \frac{1}{n_j} \sum_{i} w_i \times T_{i,j} \tag{4.1}$$

Where, w_i is the trust weight for $reviewer_i$, $T_{i,j}$ is $reviewer_i$'s rating of release R_j, and n_j is the total number of known ratings for release R_j. Each epoch (e.g., week), an agent evaluates and installs release R_{new} that has the maximal aggregated rating:

$$R_{new} = R_{argmax_j Q(j)}$$

For example, consider a system that has the ratings shown in Table 4.1. Different reviewers can have different ratings for the same version of software, as in the case of $Version_1$ (rated at 58% by $Reviewer_1$, and 77% by $Reviewer_3$)

Table 4.1: Global ratings

Reviewer	Software version	Rating%
$Reviewer_1$	$Version_1$	58%
$Reviewer_1$	$Version_2$	82%
$Reviewer_2$	$Version_2$	98%
$Reviewer_2$	$Version_3$	63%
$Reviewer_3$	$Version_1$	77%

Assuming peer x has the trust weights shown in Table 4.2. Peer x will calculate (using Equation 4.1) the combined version ratings shown in Table 4.3, and therefore, will choose $Version_1$ to update to as it has the highest rating.

Table 4.2: Trust weights for peer x

Reviewer	Trust weight%
$Reviewer_1$	85%
$Reviewer_2$	36%
$Reviewer_3$	97%

Another peer (say, peer y) with a different view (different relative trust weights of reviewers) will calculate different ratings, and might end up choosing a different version

Table 4.3: Calculated versions ratings for peer x

Version	Calculated rating
$Version_1$	124
$Version_2$	105
$Version_3$	23

of the software in question as the highest rated. If peer y's trust weights are similar to Table 4.4, it calculates the ratings shown in Table 4.5, and chooses $Version_2$.

Table 4.4: Trust weights for peer y

Reviewer	S Trust weight%
$Reviewer_1$	57%
$Reviewer_2$	91%
$Reviewer_3$	84%

Table 4.5: Calculated versions ratings for peer y

Version	Calculated rating
$Version_1$	65
$Version_2$	104
$Version_3$	71

4.2 Updating Agent's Reviewers List

Each agent keeps a list of trusted reviewers along with their respective trust weights. The recommendations from these reviewers are used to determine the best release of software to use. Agents update their trusted reviewers based on the messages that are exchanged with others. This process potentially involves updating, adding, or removing reviewers.

In all of the following algorithms (and also in the base line MRFA), agents choose the reviewers with the highest ratings. For example, if an agent has a reviewers list similar to the one shown in Table 4.2, and receives the metarecommendation message m_1 = $\langle Reviewer_2, 84\% \rangle$, it will update the entry for that reviewer accordingly. The resulting (updated) reviewers list is shown in Table 4.6

Table 4.6: Updated reviewers trust weights for peer x

Reviewer	Trust weight%
$Reviewer_1$	85%
$Reviewer_2$	84%
$Reviewer_3$	97%

However, a message announcing a reviewer with a trust weight less than the current weight will simply be ignored. For example, the message m_2 = $\langle Reviewer_1, 78\% \rangle$ will not change anything in the agent's list as $Reviewer_1$ is already known, and has a trust weight of 85%

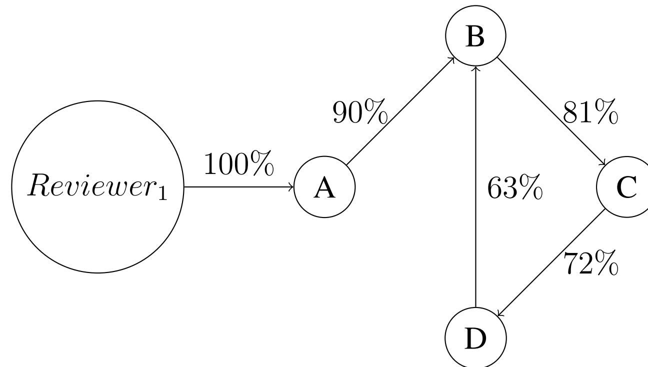


Figure 4.1: Avoiding recommendation loops

Keeping only the highest weight metarecommendation for a reviewer. Nodes A..D represent peers

Agents keep and forward only the metarecommendation with the highest rating for any reviewer to avoid redundancy and to minimize required network bandwidth. For example, agent B in Figure 4.1 knows about $Reviewer_1$ in two different ways, but it only keeps the metarecommendation from agent A (with 90% weight) and ignores the one from D (63%) as they are both advertising the same reviewer. Furthermore, to improve efficiency, agents limit the number of used/forwarded reviewers to keep the number of update messages manageable.

4.2.1 Reviewers Joining the System

When a new reviewer joins the system, that is achieved through some agent that recommends it. The introducing agent announces the availability of this new reviewer, and their initial trust weight. Receiving agents relay it to their connected peers, and so on. This approach is used by all algorithms described below.

For example, when receiving the message $m_3 = \langle Reviewer_4, 65\% \rangle$, the agent adds this newly discovered reviewer to their trusted list. In the previous example, this results in the updated list shown in Table 4.7.

Table 4.7: Adding a new reviewer

Reviewer	Trust weight%
$Reviewer_1$	85%
$Reviewer_2$	84%
$Reviewer_3$	97%
$Reviewer_4$	65%

4.2.2 Reviewers Leaving the System

Reviewers information can be discarded from the system for a variety of reasons; Some reviewers might become disconnected from the network (*off-line*), or they might not update their ratings for a long time. MRF provides a preliminary mechanism for keeping only the highest-rated reviewers, but our proposed algorithms provide a direct and more controllable way of removing outdated reviewer information.

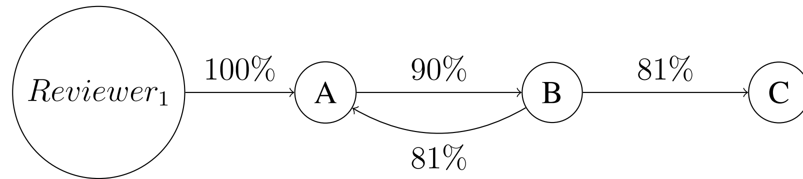


Figure 4.2: Example of metarecommendation message exchange

If not handled properly, deleting reviewers can pose serious challenges where agents might keep exchanging information about non-existent (deleted) reviewers. In the following example (Figure 4.2), agent A trusts $Reviewer_1$ 100% (directly connected) and introduces it to its neighbor, agent B (assuming an amortization factor of 90%, B's trust weight for this reviewer is now 90%). Agent B then relays this reviewer to its connected peers (agent C in this example). The message broadcast by B looks like $m = \langle Reviewer_1, 81\% \rangle$. These messages are sent to all agent B's connected peers, including agent A, which ignores them as $Reviewer_1$ is already known and has a trust weight of 100%, higher than the advertised 81%. The three agents trusted reviewers lists are shown in Table 4.8

If, at some point, agent A detects that $Reviewer_1$ is no longer available, and should not be used anymore, it drops it from its trusted reviewers list (Table 4.9). However, agent B

will keep broadcasting the messages $m = \langle Reviewer_1, 81\% \rangle$ to everyone. This time, agent A will receive the message from agent B, and adds it back to its list assuming it is a new reviewer as shown in Table 4.10.

Variations of this feedback loop can go many levels deep, and are difficult to detect or mitigate. Messages pertaining to deleted reviewers are not only a waste of network bandwidth and processing resources, but they also pose security risks to the system integrity.

Table 4.8: Trusted reviewer lists for agents A (left), B (middle), and C (right)

Reviewer	Trust%
$Reviewer_1$	100%
$Reviewer_2$	84%

Reviewer	Trust%
$Reviewer_1$	90%
$Reviewer_3$	65%

Reviewer	Trust%
$Reviewer_1$	81%
$Reviewer_4$	79%

Table 4.9: Trusted reviewer lists for agents A (left), B (middle), and C (right)

Reviewer	Trust%
$Reviewer_2$	84%

Reviewer	Trust%
$Reviewer_1$	90%
$Reviewer_3$	65%

Reviewer	Trust%
$Reviewer_1$	81%
$Reviewer_4$	79%

Table 4.10: Trusted reviewer lists for agents A (left), B (middle), and C (right)

Reviewer	Trust%
$*Reviewer_1*$	81%
$Reviewer_2$	84%

Reviewer	Trust%
$Reviewer_1$	90%
$Reviewer_3$	65%

Reviewer	Trust%
$Reviewer_1$	81%
$Reviewer_4$	79%

4.3 Algorithms to Handle Churning

In the following, we introduce new meta-recommendation algorithms based on the aforementioned ideas. The rest of the message handling procedures are similar to MRF. We use the following notations:

- $\mathcal{T}$: the set of trusted reviewers used by an agent, with their respective trust weights. $\mathcal{T} = \{\langle reviewer_i, rating_i \rangle\}_i$
- n: maximum allowed number of agent's trusted reviewers
- $\mathcal{M}$: set of incoming meta-recommendation messages $\mathcal{M} = \{\langle reviewer_i, rating_i \rangle\}_i$
- Γ: maximum life-time for meta-recommendations
- $ageOf(x)$: function that returns the age of message x (in epochs) since inception, based on additional meta-data in messages not described here for simplicity
- $sourceOf(x)$: meta-data-based function that returns the source of message x (the peer ID who sent the message)

4.3.1 Drop/remove Messages

Algorithm 1, in addition to the basic type of messages used by the baseline MRF to announce/add a new reviewer, uses a second type of messages that tell peers to drop (stop using and forwarding) a certain reviewer.

Experimentation shows this approach to be effective (described later in Section 5.5). This new type of messages is not forwarded based on *fitness* as basic recommendation messages, therefore it requires a different way to propagate it which can have a significant impact on system complexity.

For every new message received by an agent, the type of operation this message represents is first checked.

- If the type is REMOVE_MESSAGE, the agent removes the relevant reviewer indicated in this message from their trusted list (line 5 in Algorithm 1)

- Otherwise, it is a normal message announces the discovery of a reviewer by a connected peer.

  - If the reviewer is already known, its trust weight gets updated (i.e., the existing entry for this reviewer gets replaced with the newly received entry), if the received message:

    * Is coming from the original source

    * Has higher weight

    This case is represented by lines 9-22

  - If the reviewer is not already known, the agent adds it to its tentative trusted reviewers (line 26)

Finally, the agent "trims" its trusted list to keep only the highest N trusted reviewers, and forwards only those to connected peers for them to process in the next round.

4.3.2 Limited-Life Meta-Recommendations Algorithm (LLMRA)

With the method shown in Algorithm 2, performed once each *epoch*, we propose to set a maximum allowed life interval Γ for meta-recommendation data. This automatically removes any old meta-recommendations from the system after they expire, but also requires

Algorithm 1: Remove messages: Updating trusted reviewers list by peers

```
   Data: msgQueue: the incoming messages queue
   Result: Updated list of trusted reviewers
1  while msgQueue is not empty do
2      retrieve message from msgQueue
3      if typeOf(message) is REMOVE_MESSAGE then
5          list.remove(reviewer)
6      else
7          found ← false
9          if the reviewer already exists then
11             if message is from the original source then
13                 update the corresponding reviewer's weight
14             else
16                 if new message has higher weight then
18                     replace the existing reviewer with the new (received) entry
19                 end
20             end
22             found ← true
23         end
24         if not found then
26             list.append (reviewer_i)
27         end
28     end
29 end
30
32 list.sort()
34 list.trimSize(N)
35
37 forwardToConnectedPeers(list)
```

Table 4.11: Agent view using LLMRA

Reviewer	Rating%	Source	Round#
$Reviewer_1$	58%	$peer_{53}$	104
$Reviewer_2$	63%	$peer_8$	13
$Reviewer_3$	77%	$peer_{17}$	27
$Reviewer_4$	77%	$peer_{26}$	92

the reviewers to re-broadcast their reviews every α epochs, where $\alpha \leq \Gamma$. This algorithm is also referred to as *Message-Aging* in this work.

Line 7 updates trust weights of known reviewers, and Line 11 handles the addition of new reviewers. The meta-recommendations' age is then checked in Lines 15-19 and removed if expired. All known reviewers are then sorted in descending order based on their trust weights (Line 23), and truncated to the required size n, if needed (Lines 25, 27). $\mathcal{T}^*$ is then sent to all connected peers.

An example of an agent's view using this algorithm can be seen in Table 4.11

The Round# column in Table 4.11 represents the time when the corresponding reviewer was learned about or when it was last updated. If the "age" of a particular recommendation exceeds the maximum time allowed by the system, it gets dropped from the table.

The choice of Γ (the maximum allowed age for meta-recommendation messages) is a significant factor in the overall system performance. A smaller value of Γ (a short message life interval) requires agents to exchange frequent updates, which can consume network bandwidth and processing power, but minimizes the time required for agents to detect and delete off-line reviewers. On the other hand, a large Γ value allows messages to exist for a

longer time in the system which reduces the required frequency of updates, but delays the discovery of the aforementioned situations.

4.3.3 Non-positive Meta-Recommendations Algorithm (NPMRA)

In Algorithm 3, we use *non-positive* trust weight values to indicate the removal of a reviewer. Agents keep a separate set of these reviewers ($\mathcal{D}$ in Lines 3,29). Previous meta-recommendation operation selects only the reviewers with the highest ratings, but the addition of set $\mathcal{D}$ helps keep track of the deleted reviewers. Agents update their connected peers regarding these deleted reviewers.

If a non-positive trust weight for a certain reviewer is received, then the agent reevaluates it and may remove it from their trusted set $\mathcal{T}$ (Lines 9, 11). Line 15 handles any updates in reviewers already known to the agent. If the received message announces a reviewer not yet known to the agent, then it is added to the trusted set $\mathcal{T}$ (Line 20). All deleted reviewers are added to set $\mathcal{D}$ (Line 25), which is then forwarded to all connected peers, along with $\mathcal{T}$.

4.3.4 Update-driven Meta-Recommendations (UDMRA):

In Algorithm 4.3.4, agents keep a "view" of all the reviewers they know about (kept permanently). Agents calculate changes since last update messages sent out (the *delta*) and send only these changes to all connected peers except the sources of these changes to avoid meta-recommendation loops. The receiving agents update their trust weights of reviewers if they receive an update from the same source that informed them about that reviewer in

Algorithm 2: LLMRA: updating agent's reviewers list

```
1  LLMRA (𝓜)
       input  : set of incoming meta-recommendation messages
       output: 𝓣*, updated trusted reviewers list
3      foreach ⟨E_i, R_i⟩ ∈ 𝓜 do
5          if ∃x, ⟨E_i, x⟩ ∈ 𝓣 then
7              𝓣 ← (𝓣 \ ⟨E_i, x⟩) ∪ ⟨E_i, R_i⟩
9          else
11             𝓣 ← 𝓣 ∪ ⟨E_i, R_i⟩
12         end
13     end
15     foreach t_i ∈ 𝓣 do
17         if ageOf(t_i) > Γ then
19             𝓣 ← 𝓣 \ t_i
20         end
21     end
23     𝓣* ← (𝓣) sorted into descending order of rating
25     if |𝓣*| > n then
27         𝓣* ← first (highest rated) n items of 𝓣*
28     end
30     return 𝓣*
```

Algorithm 3: NPMRA: updating agent's reviewers set

```
1  NonPositive (M)
     input  : set of incoming meta-recommendation messages
     output: T, updated trusted reviewers set
     output: D, set of removed reviewers
3    D ← ∅
5    foreach ⟨E_i, R_i⟩ ∈ M do
7      if ∃x, ⟨E_i, x⟩ ∈ T then
9        if R_i < 0 and reviewer reevaluation negative then
11         T ← T \ ⟨E_i, x⟩
13       else
15         T ← (T \ ⟨E_i, x⟩) ∪ ⟨E_i, R_i⟩
16       end
18     else
20       T ← T ∪ ⟨E_i, R_i⟩
21     end
23     if reviewer E_i removed then
25       D ← D ∪ ⟨E_i, R_i⟩
26     end
27   end
29   return T, D
```

the first place. Otherwise, the trust weight is only updated if a higher weight is received for a specific reviewer (higher than the one already known by the agent)

Preliminary experiment results show this to be the most efficient algorithms as it requires the least number of recommendation messages. However, because of time limits, these results are not included in Chapter 5 because of lack of time.

Reducing the amount of information to be exchanged between agents, and therefore, the processing power required by each agent, opens doors to new application where network bandwidth or processing power might be limited. On the other hand, as agents need to keep a full "view" of connected reviewers at all times, this algorithm might not be suitable for applications where memory is limited (for example, in embedded systems, or IOT devices).

While the agents' view in this algorithm is stored in a similar way to what was shown in Table 4.11, it applies a different policy to updating the entries in the view. Received updates are stored in the view only in one of two cases (as shown in Line 4 of Algorithm 21):

- The message is coming from the same peer that informed this agent about that particular reviewer in the first place.
- The incoming message carries a higher trust weight than the current entry stored in the view.

When updating connected peers using UDMRA (say, at round i), agents construct and send out a *change list* (Δ_i) that contains the following, only if they have not been included into a previous Δ_j, where $j < i$

- New reviewers added to the current view.

- Reviewers removed from the current view.

- Updates in trust weights of reviewers in the current view.

When receiving updates, agents adds or updates metarecommendation messages only if they come from the same source that announced the reviewer, or if they carry a higher trust weight higher than the one known to this agent so far. This helps avoid recommendation cycles, where agent keep exchanging messages about non-existing reviewers.

Algorithm 4: UDMRA: Outgoing updates

```
1  UDMRAsend (M)
     input  : set of incoming meta-recommendation messages, each in the form
              ⟨reviewer_i, rating_i⟩
     output: T, updated trusted reviewers set
3    Δ ← ∅
4    foreach m ∈ messages do
5      if (m has a higher trust weight) or (m is from the same known source))
        then
6        update the entry for m
7        Δ ← Δ ∪ m
8    end
9    messages ← ∅
10   foreach a ∈ peers do
11     foreach m ∈ Δ do
12       if (!(sourceOf(m) == a)) then
13         g ← m
14         if (ratingOf(g) < 0) then
15           ratingOf(g) ← 0;
16         sourceOf(g) ← this
17         send message g to peer a
18     end
19   end
21 return T
```

Algorithm 5: UDMRA: Incoming updates

```
1  UDMRAreceive (M)
     input  : incoming meta-recommendation message m
     output: T, updated trusted reviewers set
2    receive(Msg m)
4    if (m is from the same known source) or weightOf(m)) > previous weight) then
5      messages ← messages ∪ m
6    end
```

Chapter 5

Experimentation

We ran the experiments on a total of 10,000 peers ($P_0..P_{9999}$) divided into 100 neighborhoods, each neighborhood containing 100 peers (N_0 contains peers $P_0..P_{99}$, N_1 contains peers $P_{100}..P_{199}$, etc.).

Each peer is maintaining lists with a maximum of 12 reviewers. Each peer is linked to 50 other peers; 48 being within the same neighborhood as the peer, and the other two links are to peers in other neighborhoods. A set of 100 reviewers ($R_0..R_{99}$) are introduced manually at time T_0 by the selected set of sophisticated peers (Reviewer R_i is introduced by peer P_{i*100} with weight = 100%). The unsophisticated peers, who do not manually select their trusted reviewers, build their list of trusted reviewers based on the messages received from connected peers. Those recommendations are evaluated daily at a given UTC time and the decision is then disseminated to peers. An amortization factor of 0.95 is used in all experiments.

5.1 Computational Environment

Experiments were ran on an IBM iDataplex system having 1,720 processor cores in 63 compute nodes and 4.4TB of RAM. The system runs CentOS 7 Linux.

5.2 Simulator

The main simulation model used in all experiments is implemented in Java 8. The code will be made publicly available on GitHub.

5.3 Problem Generator

The problem generator was also implemented in Java 8. It creates the starting state of the network based on the parameters provided in a JSON file. This initial state is then stored in another JSON that is used by the main simulator.

5.4 Parameters

The experiments simulate 100 days (rounds). Note that the updates can be weekly rather than daily, and therefore the simulated time scales accordingly. The simulator guarantees that messages are sent in random order to one's peers, to avoid structural bias. The experiments were performed 10 times, and the results shown below are averaged over these 10 results.

We run simulations to perform all the experiments, using the following parameters: All experiments use the following parameters:

- Each experiment is run for 100 rounds ($t_0, t_1, .., t_{99}$). The rounds can represent any fixed period of time (daily, weekly, and so on).
- Every experiment was performed 20 times with different random seeds, and the results were averaged over all runs
- 10,000 peers ($P_0, P_1, .., P_{9999}$) grouped in 100 neighborhoods ($N_0, N_1, .., N_{99}$). Neighborhood N_i contains peers $P_{i*100}..P_{((i+1)*100)-1}$
- The maximum number of trusted reviewers kept by each peer is 12
- Each peer establishes 50 outgoing connections randomly chosen at t_0. 48 of these connections are made to peers within the same neighborhood, and the other 2 connections are made to peers in different neighborhoods.
- All experiments use an amortization factor $\lambda = 0.95$
- The escape function S_e uses the $Q(j)$ quality valuation
- Reviewer reevaluation for deletion tests meta-recommendation source
- 10 software releases (distros) are simulated ($release_1$ .. $release_{10}$), two of which are considered compromised (namely, $release_4$ and $release_{10}$).
- An agent is considered a *casualty* if it adopts one of the compromised releases.

- 100 *reviewers* are introduced ($reviewer_i$ is introduced by a random agent in $neighborhood_i$).
- A *epoch* refers to the period of time between message exchanges among agents (can be a day, a week, ..etc.)
- The *attackers* refers to the reviewers who recommend compromised releases of the software to other agents, possibly due to be taken over by other attackers.
- Simulations ran on a cluster system comprised of 63 compute nodes (Total of 1,720 processor cores and 4,397GB RAM) under CentOS 7 Linux.
- The code is made available.

We use 10 different versions (releases) of the software (R_0, R_1, ..., R_9), and each reviewer ranks them differently. The set of trusted reviewers (and their trust weights) for each peer determines which version to adopt.

In general, the parameters used in these experiments are similar to what is used in [6, 7]. We ran the experiments on a total of 10,000 peers ($P_0..P_{9999}$) divided into 100 neighborhoods, each neighborhood containing 100 peers (N_0 contains peers $P_0..P_{99}$, N_1 contains peers $P_{100}..P_{199}$, etc.).

5.5 Experiments

We first introduced all 100 *reviewers* at time T_0, and observe their usage distribution after 50 rounds. The resulting distribution (shown in Figure 5.7) serves as a baseline, and it has

the following characteristics: Mean= 1200.00, Median= 1104.00, Mode= 798.00, Range= 1865.00, Std Dev.= 445.35

Figure 5.4 shows a comparison between the proposed *message time-out* approach (the dashed red line) and the base-line approach (the dotted blue line) in terms of number of peers (shown on the *y-axis*) that use the deleted reviewer for each round (*x-axis*).

Since early reviewers in current recommenders tend to have a higher user base, attacks on early reviewers are more efficient. Later reviewers have less opportunities to overcome them. This can be seen as a drawback for the current dissemination mechanism. On the other side, this illustrates a natural effect of the desired behavior of taming switches between reviewers used by a given user, a heuristic introduced for limiting the number of casualties.

The obtained trade-off between casualties due to switching and casualties that can be due to rigid adoption of original reviewers may require adjustment based on perceived threats. Ideally this adjustment can be the result of an automatic negotiation process based on individual user choices, to be addressed in future research.

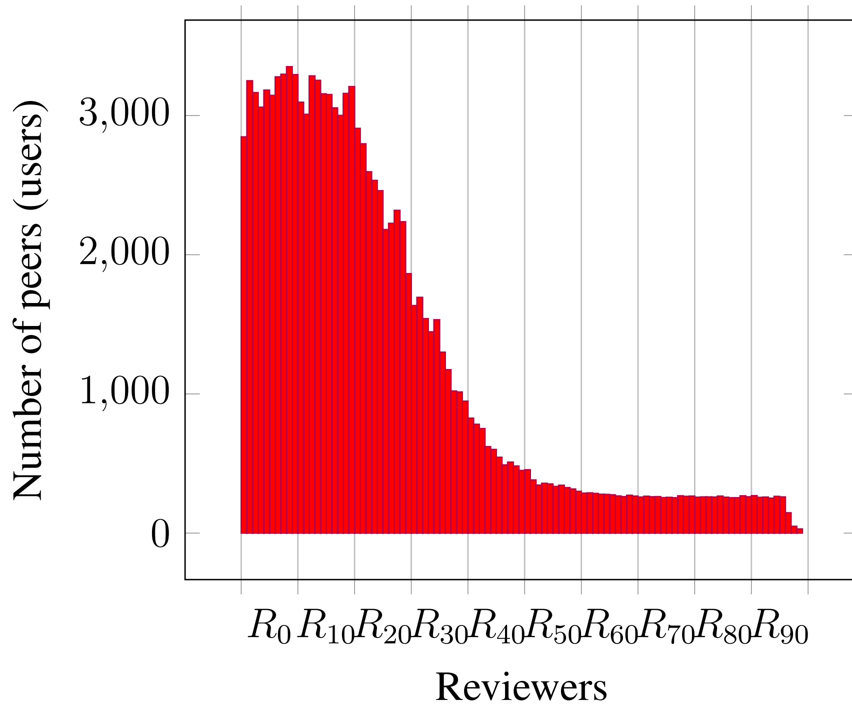


Figure 5.1: Adding reviewers: equal initial weights (all 100%)

In the first experiment, we introduce one reviewer at each time step (Reviewer R_i is introduced by peer P_{i*100} at time t_i). We observe the resulting usage distribution (shown in Figures 5.1, 5.2). If all reviewers are introduced with the same initial weight (100%), reviewers usage by peers increases with the period of time they exist in the system (the earlier they are introduced) as shown in Figure 5.1. If they are introduced with random initial weights (between 0%-100%), only those with higher weights will eventually dominate the distribution (shown in Figure 5.2).

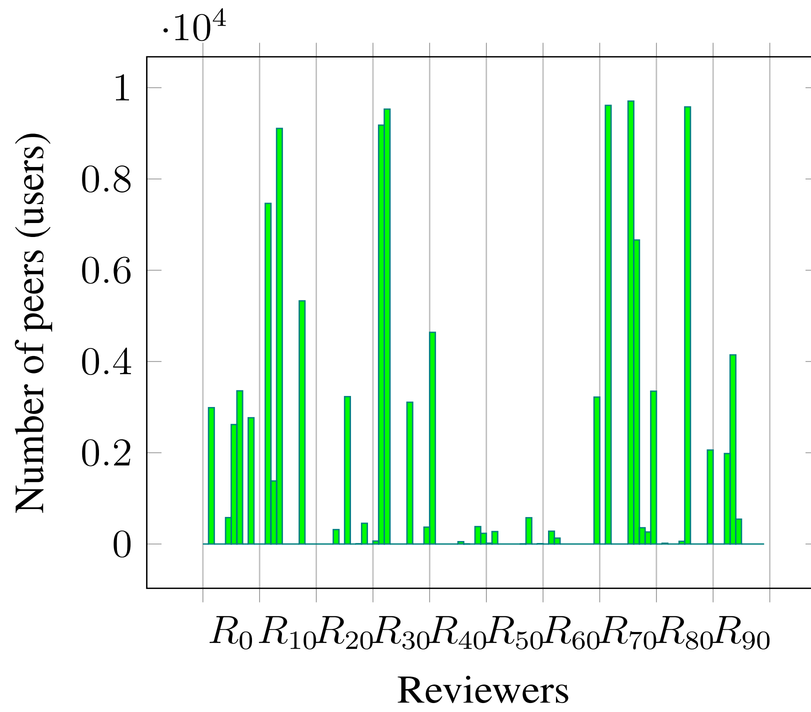


Figure 5.2: Adding reviewers: random initial weights

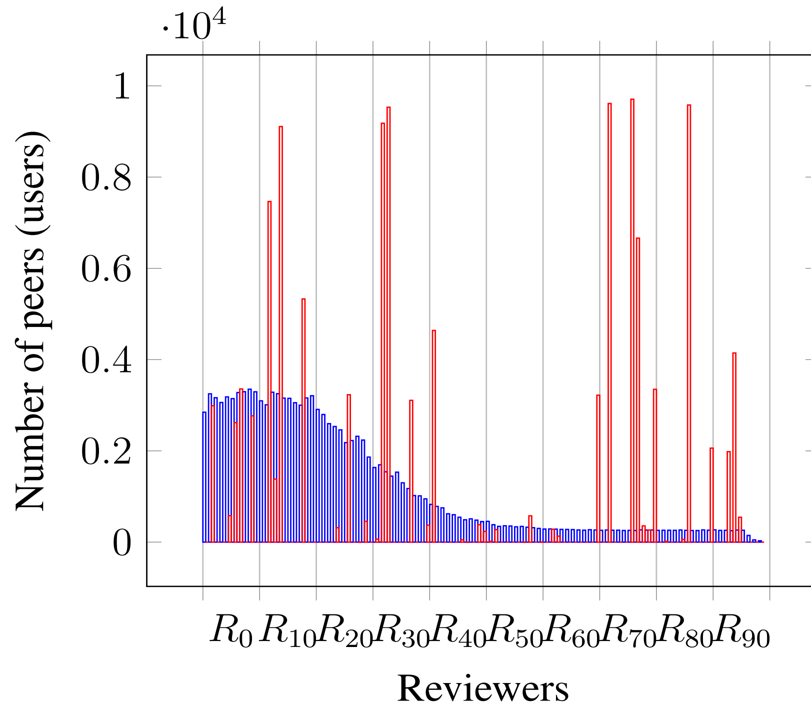


Figure 5.3: Adding reviewers: equal vs. random weights

We compare some of the proposed approaches to handle removing reviewers. In all cases, reviewer R_{59} is removed at t_{41}. The approaches are evaluated based on the amount of time it takes from the time the reviewer is removed, until the time all peers stop using it (its usage becomes 0). Failing to reach zero usage (as in the case with the base-line algorithm) indicates that the approach is not effective. The results are shown in Figure 5.4.

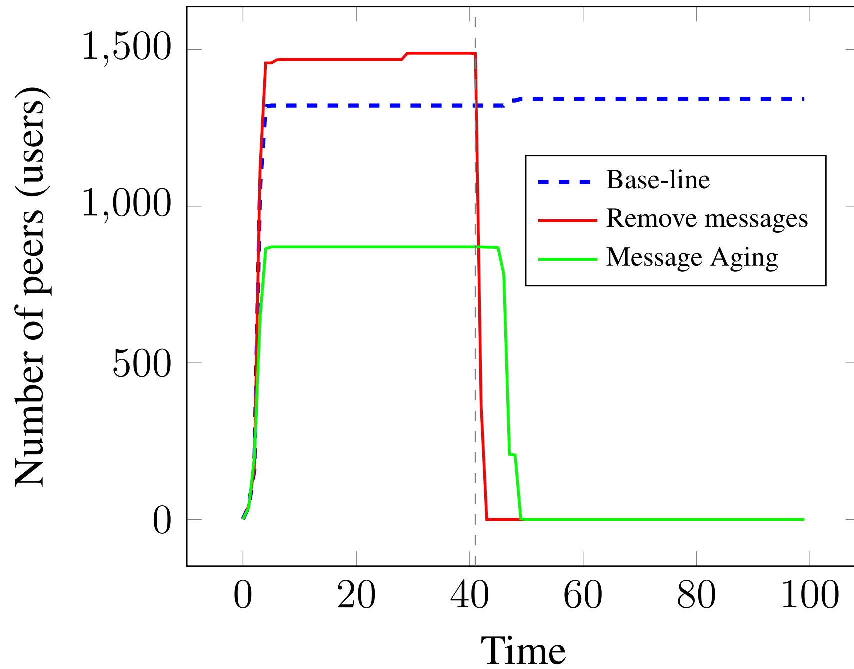


Figure 5.4: Removing a reviewer: comparing approaches

We show in Figure 5.4 how the system reacts to removing reviewers using the Message-Aging approach (all introduced at t0 , and using timeout = 4). Reviewer R55 is removed at t5 , R66 is removed at t73 , and R88 is removed at t28 . The black dotted line shows the number of remove operations performed by the system (by all peers combined). The base line algorithm referenced here is MRF.

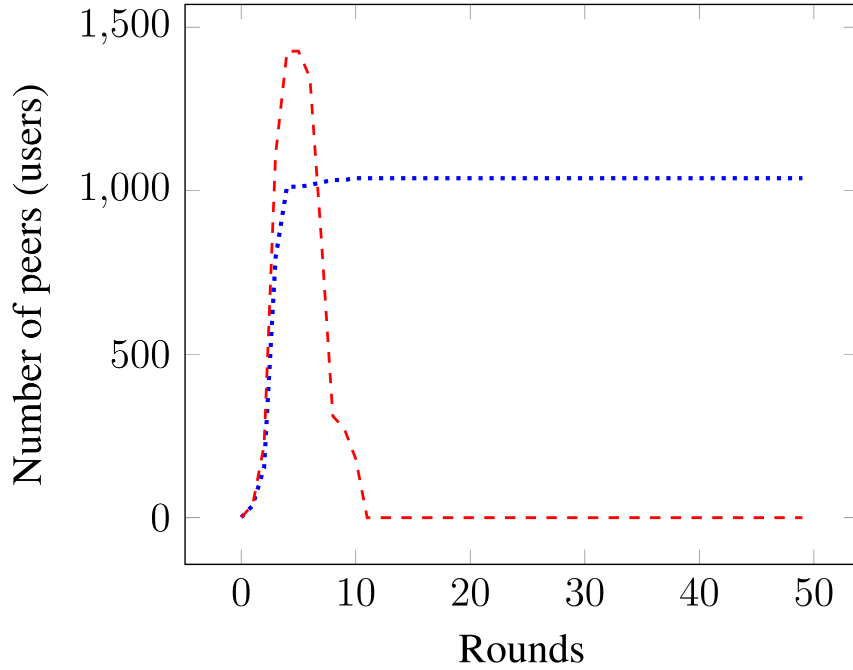


Figure 5.5: Deleting a reviewer

Usage by peers: comparing base-line to message time-out approach (red dashed line)

Figure 5.5 shows a comparison between the proposed *message time-out* approach (the dashed red line) and the base-line approach (the dotted blue line) in terms of number of peers (shown on the *y-axis*) that use the deleted reviewer for each round (*x-axis*). In this experiment, reviewer R_{59} is removed from the system at round 3, using a time-out of 4. The results show that the deleted reviewer is being used by a lesser number of peers with every round (from round 3 to 10), until it is removed completely at round 11 (it takes twice the *time-out* to achieve complete removal).

The 100 reviewers ($R_0..R_{99}$) are introduced manually at time t_0 by peers (Reviewer R_i is introduced by peer P_{i*100} with weight = 100%). Figure 5.7 shows the total number of

operations performed globally in the system (by all peers). The system converges at t_{14}. Figure 5.7 shows the distribution of reviewers usage by peers after 100 rounds.

The 100 reviewers ($R_0..R_{99}$) are introduced manually at time t_0 by peers (Reviewer R_i is introduced by peer P_{i*100} with weight = 100%). Figure 5.6 shows the total number of operations performed globally in the system (by all peers). The system converges around t_{14}. Figure 5.7 shows the distribution of reviewers usage by peers after 100 rounds.

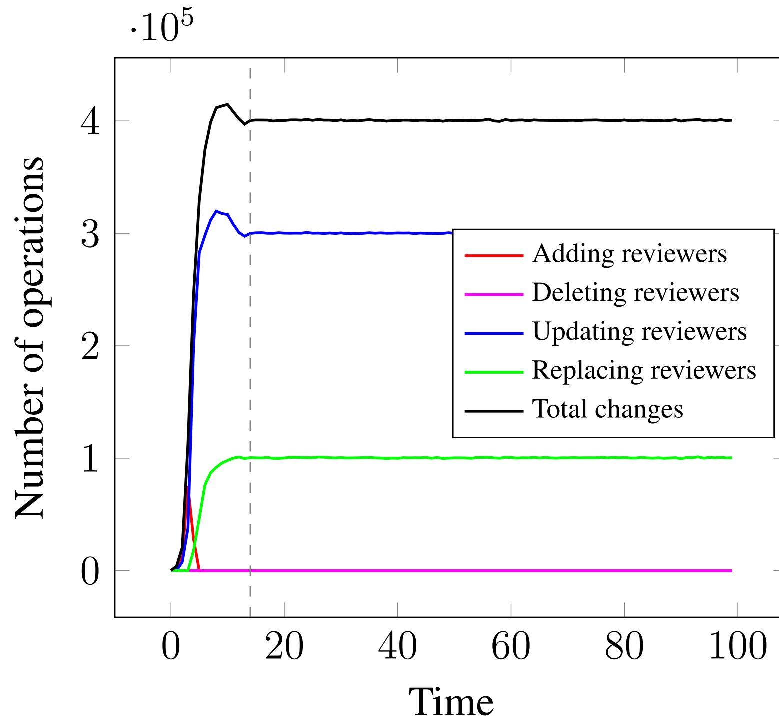


Figure 5.6: Number of operations performed by all peers (combined) per time

In the following experiment, we introduce one reviewer at each time step (Reviewer R_i is introduced by peer P_{i*100} at time t_i). We observe the resulting usage distribution (shown in Figure 5.3). If all reviewers are introduced with the same initial weight (100%), reviewers usage by peers increases with the period of time they exist in the system (the early

they are introduced) as shown by the blue bars. If they are introduced with random initial weights (between 0%-100%), those with higher weights will dominate the distribution.

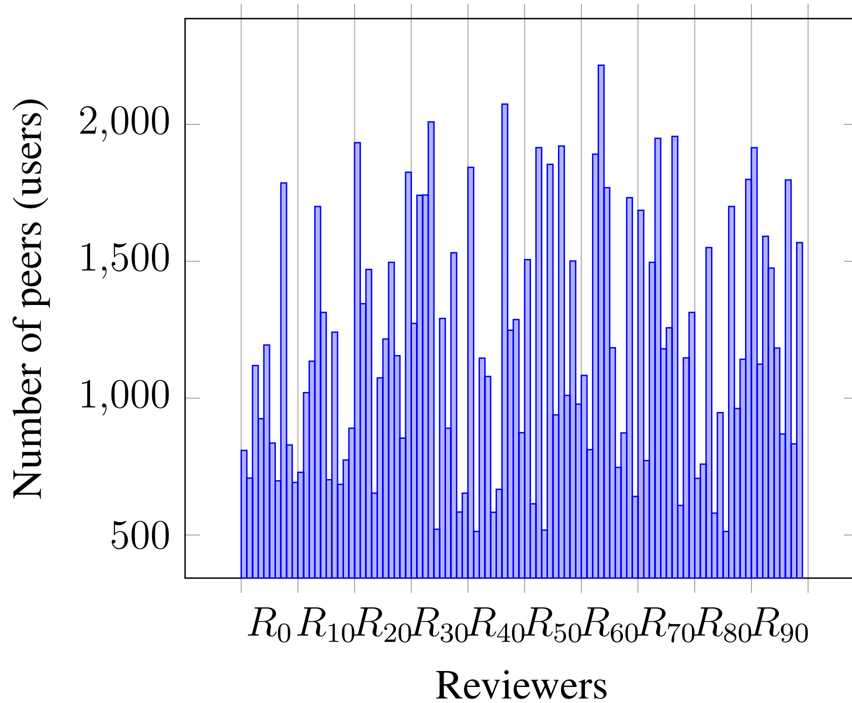


Figure 5.7: Reviewer usage by peers

We show in Figure 5.8 how the system reacts to removing reviewers using the *Message-Aging* approach (all introduced at t_0, and using $timeout = 4$). Reviewer R_{55} is removed at t_5, R_{66} is removed at t_{73}, and R_{88} is removed at t_{28}. The black dotted line shows the number of *remove* operations performed by the system (by all peers combined).

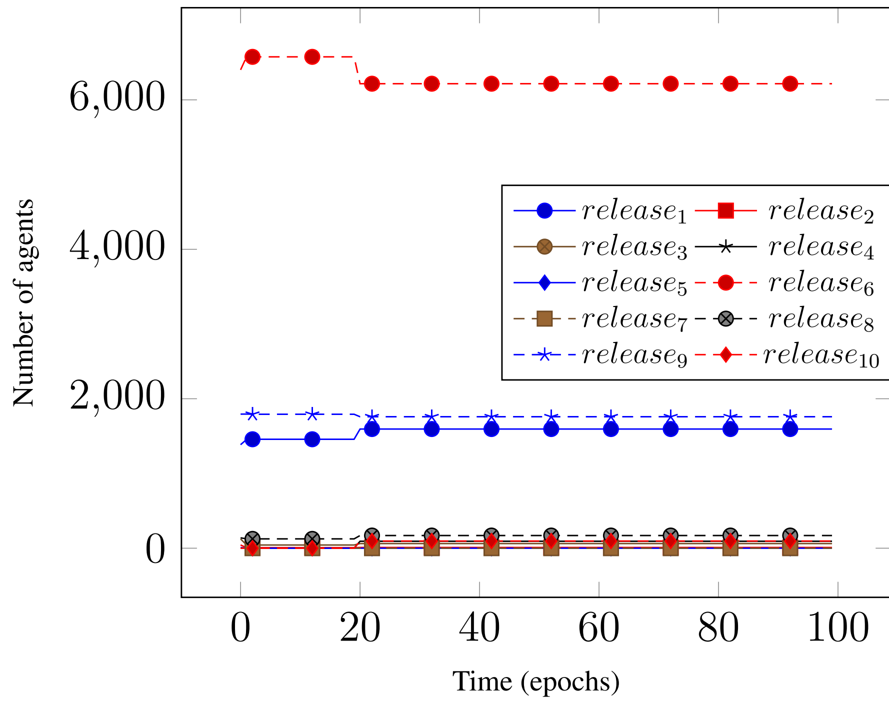


Figure 5.9: NPMRA: release usage over time

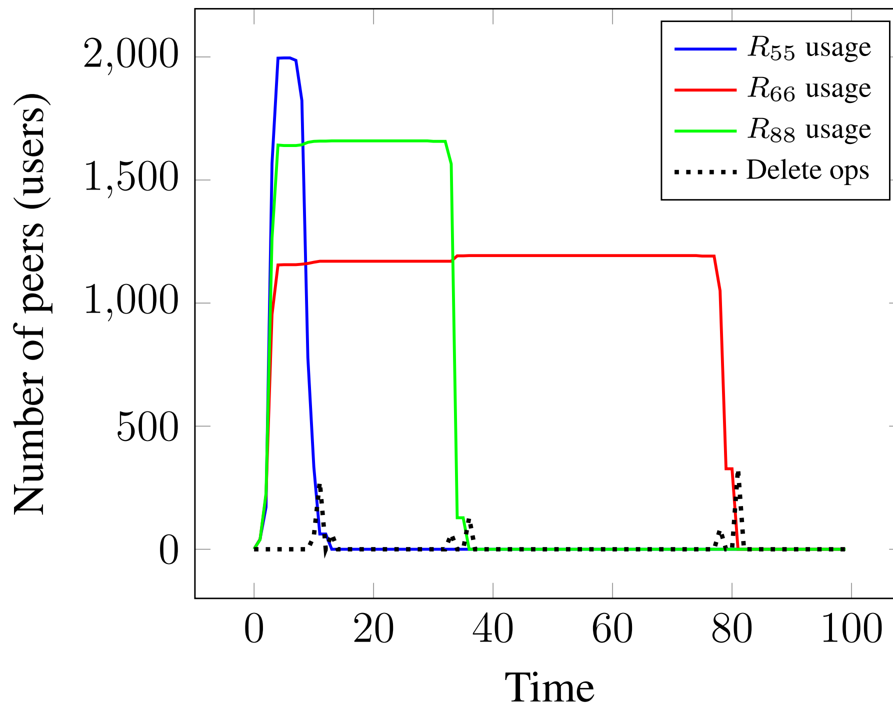


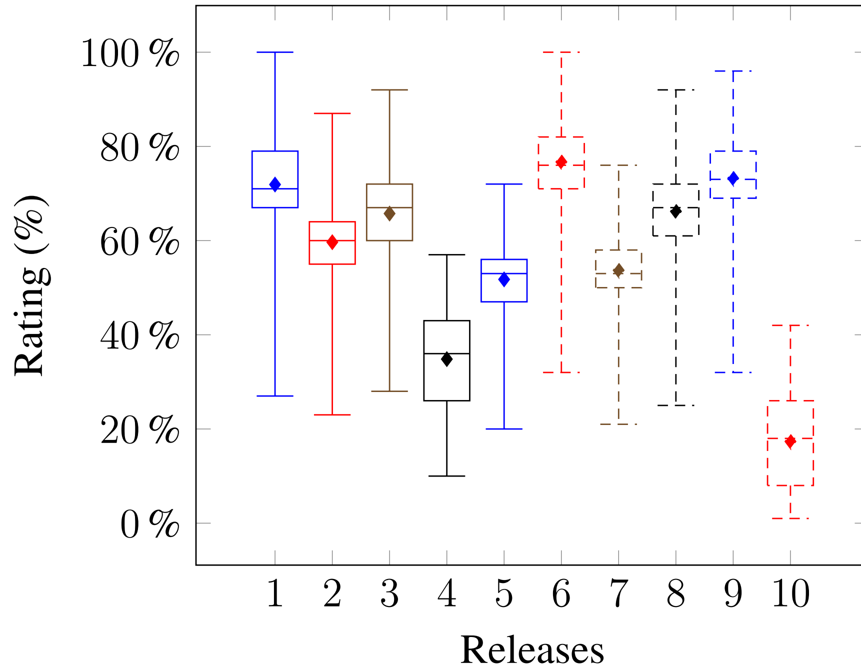
Figure 5.8: Removing reviewers: Message-Aging approach

Figure 5.9 shows the usage of different FOSS releases, with the number of agents shown on y-axis, over the duration of the simulation, and the epoch number is on the x-axis. We notice, due to the effects of the meta-recommendation being exchanged between agents, the releases that have the highest ratings in the scenario used in this experiment (like $Release_6$), have higher usage by agents. On the other hand, low-rated releases (like $Release_4$, and $Release_{10}$), are used by very few agents.

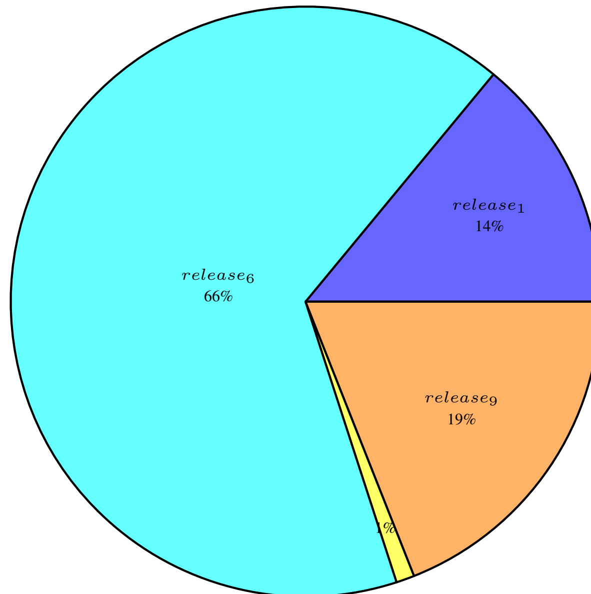
We also notice that the system converges relatively quickly, shown by the lack of any steep inclines or declines in the usages of the releases. Even after introducing changes of ratings over the course of the simulation, the system responds and stabilizes within a few epochs. In this experiment, we decreased the rating of $Release_6$ by a small amount (2%) at $epoch_{18}$, which made some agents switch to $Release_1$ (shows an increase of usage around the same time frame). LLMRA showed similar results (not shown here).

Figure 5.10 gives two more different views of the distribution (usage) of the different releases. Figure 5.10-a shows the key metrics of the over-all release ratings exchanged by all agents during all epochs combined, and Figure 5.10-b shows the final distribution (at last epoch) of the percentages of agents that ended up using each release. This shows the desired effect of the best releases (highly rated by reviewers) being used more commonly than the lowly rated releases (possibly compromised). For example, 66% of users adopt $Release_6$ compared to 0% for $Release_4$.

In Figure 5.11, we look at the number of *casualties* resulting from different *amortization factors* λ (values used are 0.25, 0.5, 0.8, and 0.95) for different percentages of



(a) NPMRA: key descriptors of releases over-all ratings



(b) NPMRA: release usage by agents at last epoch

Figure 5.10: NPMRA: releases usage and over-all ratings

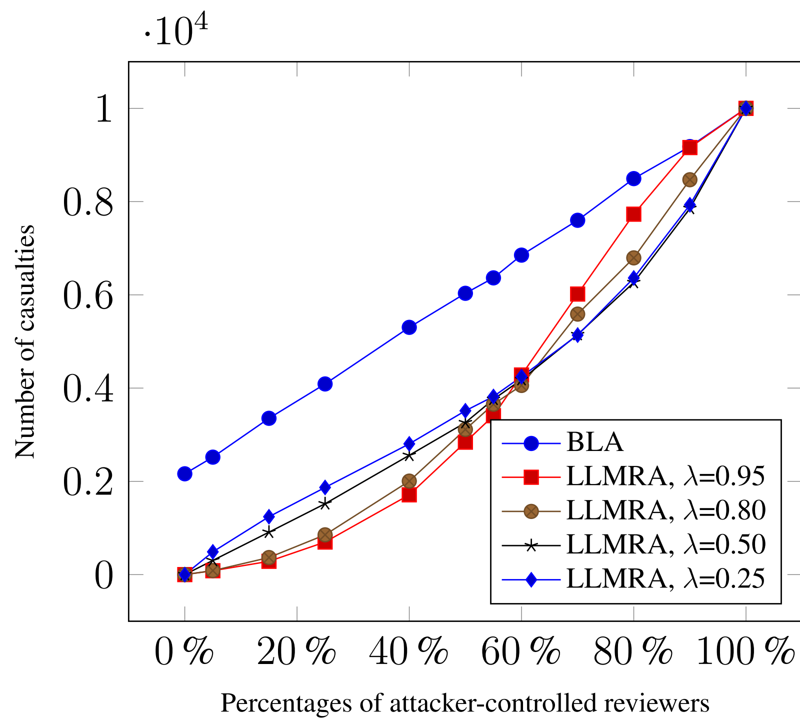


Figure 5.11: LLMRA casualties
Casualties for different amortization factors and different attacker-controlled reviewers percentages

attacker-controlled reviewers ranging from 0% to 100%. The casualties with BLA (randomly picking one of the available releases) is also included in this graph for comparison.

Figure 5.11 shows that when 100% of the reviewers are controlled by attackers, all agents in the system become casualties to the attack, regardless of the algorithm used. However, for any lower percentage of attacker-controlled reviewers, it is more beneficial to use any of the meta-recommendation variations. Another observation we make here is that using a higher amortization factor λ can be more beneficial if the percentage of attackers is low, but for higher attacker percentages, it is better to have a lower value for the amortization factor. This observation is due to the amortization factor having a direct effect on propagating trust in the whole system, so higher λ values will have a higher impact on the minority (good if attackers are a minority, bad if "good" reviewers are the minority)

For all tested values of λ, LLMRA performs better (in terms of lower number of casualties) than BLA. NPMRA results were very similar to LLMRA, and were not included.

In this experiment, we measure the impact of adding and removing reviewers on the number of agents using them, as a function of the chosen meta-recommendation algorithm. Figure 5.12 shows the numbers of agents (y-axis) using reviewers over time (epochs) on x-axis. For readability, only representative reviewers are shown.

$Reviewer_{257}$ and $Reviewer_{279}$ have been added in $epoch_{10}$ and $epoch_{20}$ respectively in this simulation (agents usage of these reviewers increases accordingly), and $Reviewer_{32}$

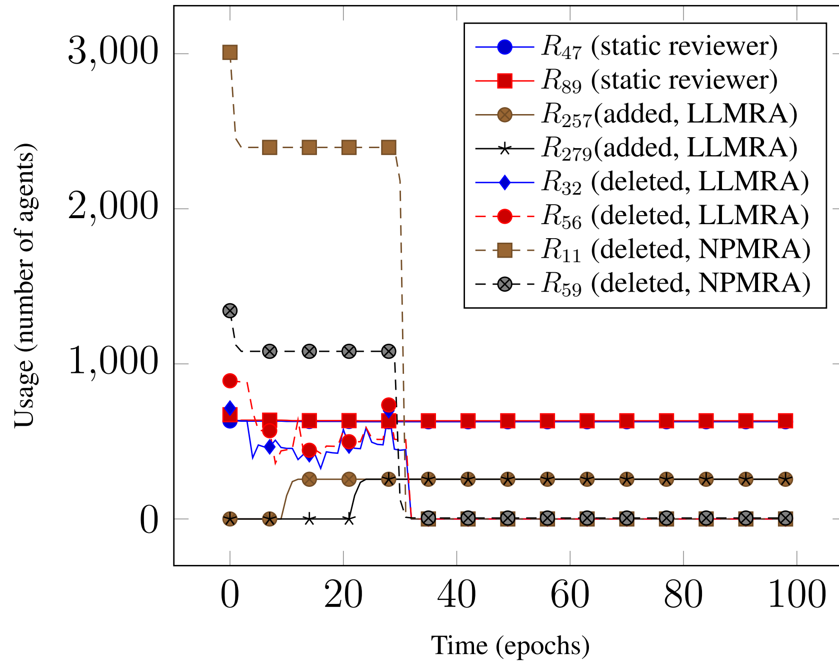


Figure 5.12: Numbers of agents using reviewers
Only selected examples shown for readability

and $Reviewer_{56}$ have been removed using LLMRA in $epoch_{30}$ (usage decreases accordingly). The repeating pattern observed here is a characteristic of LLMRA as the algorithm periodically removes expired recommendations. A similar effect is observed in Figure 5.13.

$Reviewer_{11}$ and $Reviewer_{59}$ have been removed using NPMRA in $epoch_{30}$ (their usage drops accordingly). We also show $Reviewer_{47}$ and $Reviewer_{89}$ that did not encounter any events. MRF does not show any reduction of casualties as it does not handle the addition or deletion of reviewers.

Figure 5.13 shows the effects of similar dynamic changes on the numbers of casualties in the presence of attackers. Half of attacker-controlled reviewers are removed in $epoch_{30}$ (casualties decrease accordingly), and the rest in $epoch_{60}$ (no casualties remain). It can be observed that removing attacker-controlled reviewers from the system reduces the number

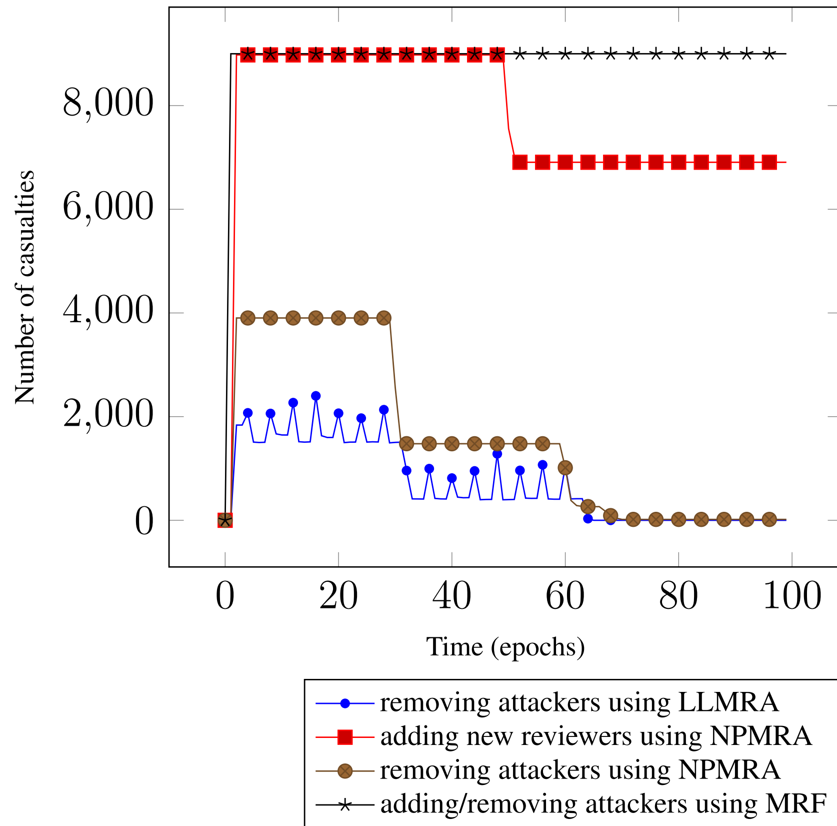


Figure 5.13: Reducing numbers of casualties

Removing attacker-controlled reviewers or by adding new reviewers.

of casualties. Number of casualties is minimized by removing all attacker-controlled reviewers. The introduction of new reviewers (20% new reviewers added in $epoch_{50}$ in this experiment) is also shown to reduce casualties as some of the casualty agents can use the recommendations from these newly added reviewers to escape the attacker-recommended release, and to choose a non-compromised release instead.

Chapter 6

Conclusions

The wide spread of free open-source software (FOSS) -encouraged by numerous success stories- attracted many developers (individuals and organizations) to utilize this model. That, however, made it a target for attackers as well, whether to promote their own versions of popular software, or to exploit vulnerabilities in the development and deployment cycle, or just to disrupt an entire system to force users to look for other alternatives that might be more beneficial for the attackers.

In this research, we propose an improved metarecommender for reviewers of updates for Free Open Source Software that can help end-users to decide whether to update to a particular version of the software or not. This framework is robust to the dynamic changes in its environment, and aims also to minimize the effects of some attacks that target reviewers (recommenders) with the goal of influencing end users decision of updating their software.

In this study, we address effects of various changes in reviewers in the meta-recommendations framework for FOSS updates. We study the effects of reviewer churning in Meta-Recommendations, and find that existing algorithms cannot take advantage of such events. We propose modifications to the messages that allow peers to maintain the up-to-date trust weight of reviewers, as determined by the source peer. We also ensure that any new reviewers joining the system can get an equal chance to be selected, by basing the selection decision solely on the trust weight of those reviewers. We show that using LLMRA or NPMRA, the number of casualties of an attack (the agents who end up using a compromised release of FOSS) can be reduced.

We investigate different approaches for removing reviewers (churning): *Self-announcement*, the use of *Zero/Negative trust weights*, introducing *Drop/remove messages*, and *Message Aging*.

Simulation results show that while the *Drop/remove messages* approach is the most effective, it requires major changes in the current framework, and most algorithms. It is also expected to increase the network traffic due to the added messages. This will be investigated further in future work, along with the effect on the scalability.

Experiments with all proposed algorithms show that agents choose releases based on their ratings. Simulations averaged over 20 instances show good resistance to coordinated attackers. The introduced recommender mechanism is shown to clean the list of reviewers within 10-15 rounds. The message-aging technique is shown to have a significantly better distribution of reviewers with respect to churning.

The experiments also show the effectiveness of proposed algorithms in reducing casualties either by adding new reviewers to the system and allowing agents to use them instead of attacker-controlled ones, or by removing compromised or outdated reviewers. Removing all attacker-controlled reviewers using LLMRA or NPMRA recovers all casualties. MRF does not show any reduction of casualties when adding or removing reviewers to the system.

All proposed algorithms have preferred traits in certain situations. LLMRA does not require any additional data structures to be maintained and uses less memory in general, but requires more frequent exchanges of meta-recommendation messages. On the other hand, NPMRA is more scalable and easier to implement.

The *Message Aging* approach is easier to incorporate in the framework, while providing an acceptable effectiveness, compared to the base-line and the *Zero/Negative trust weights* approaches. The *Zero/Negative trust weights* is considered incompatible with this framework, where messages with lower weights are most likely to be discarded.

We intend, in future work, to improve efficiency by minimizing the number of exchanged messages. We intend to utilize a distributed ledger technology such as *blockchain* to share agents' data. We also plan to apply optimized variations of proposed algorithms such as Update-Driven Metarecommendations (UDMRA) in environments with constrained resources such as embedded systems and IOT devices.

References

[1] Android 10 update issue. https://support.google.com/pixelphone/thread/15667336?hl=en.

[2] Camscanner app malware update. https://www.androidauthority.com/camscanner-malware-play-store-1023576/.

[3] Google chrome white screen of death bug (2019). https://www.forbes.com/sites/daveywinder/2019/11/15/google-chrome-white-screen-of-death-browsers-crash-as-google-tests-new-feature/#431804bd1188.

[4] Gediminas Adomavicius and Alexander Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE transactions on knowledge and data engineering*, 17(6):734–749, 2005.

[5] Charu C Aggarwal. Attack-resistant recommender systems. In *Recommender Systems*, pages 385–410. Springer, 2016.

[6] Khalid Alhamed, Marius C Silaghi, Ihsan Hussien, Ryan Stansifer, and Yi Yang. Stacking the deck attack on software updates: Solution by distributed recommendation of testers. In *Web Intelligence (WI-2013) Volume 02*, pages 293–300. IEEE, 2013.

[7] Khalid Alhamed, Markus Zanker, Shakre Elmane, and Marius Silaghi. P2p meta-recommenders: Aggregated diversity maximization as a bulwark against attacks on reviewers. In *2016 IEEE/WIC/ACM International Conference on Web Intelligence (WI)*, pages 208–215. IEEE, 2016.

[8] Anthony Bellissimo, John Burgess, and Kevin Fu. Secure software updates: Disappointments and new challenges. In *HotSec*, 2006.

[9] David M Berry. *Copy, rip, burn: The politics of copyleft and open source*. Pluto Press, 2008.

[10] Jürgen Bitzer and Philipp JH Schröder. *The economics of open source software development*. Elsevier Science Inc., 2006.

[11] Jesus Bobadilla, Fernando Ortega, Antonio Hernando, and Javier Alcalá. Improving collaborative filtering recommender system results and performance using genetic algorithms. *Knowledge-based systems*, 24(8):1310–1316, 2011.

[12] JesúS Bobadilla, Fernando Ortega, Antonio Hernando, and JesúS Bernal. A collaborative filtering approach to mitigate the new user cold start problem. *Knowledge-Based Systems*, 26:225–238, 2012.

[13] Robin Burke. Knowledge-based recommender systems. *Encyclopedia of library and information systems*, 69(Supplement 32):175–186, 2000.

[14] Tuesday Bwalya, Akakandelwa Akakandelwa, and Milena Dobreva-McPherson. Adoption and use of free and open source software (foss) globally: An overview and analysis of selected countries. *ZAJLIS Journal*, 3, 2019.

[15] Isabel Candal-Vicente, Segundo Castro-Gonzáles, and Janely García-Cortés. Evaluation of vulnerabilities in computer systems users. *Journal of Information System Security*, 13(1), 2017.

[16] Justin Cappos, Justin Samuel, Scott Baker, and John H Hartman. A look in the mirror: Attacks on package managers. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 565–574. ACM, 2008.

[17] Christian Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. Distributed application tamper detection via continuous software updates. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 319–328, 2012.

[18] Christof Ebert. Open source drives innovation. In *IEEE Software*, volume 24(3), pages 105–109. IEEE, 2007.

[19] Shakre Elmane and Marius Silaghi. Handling changes of update reviewers in distributed free and open-source software meta-recommendations. In *IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON)*, pages 638–644. IEEE, 2017.

[20] Zekeriya Erkin, Thijs Veugen, and Reginald L Lagendijk. Privacy-preserving recommender systems in dynamic environments. In *2013 IEEE International Workshop on Information Forensics and Security (WIFS)*, pages 61–66. IEEE, 2013.

[21] Joseph Feller and Brian Fitzgerald. A framework analysis of the open source software development paradigm. In *21st international conference on information systems (ICIS2000)*. Association for Information Systems (AIS), 2000.

[22] Karl Fogel. *Producing open source software: How to run a successful free software project*. " O'Reilly Media, Inc.", 2005.

[23] Domenic Forte, Ron Perez, Yongdae Kim, and Swarup Bhunia. Supply-chain security for cyberinfrastructure [guest editors' introduction]. *Computer*, 49(8):12–16, 2016.

[24] Sarika Jain, Anjali Grover, Praveen Singh Thakur, and Sourabh Kumar Choudhary. Trends, problems and solutions of recommender system. In *International Conference on Computing, Communication & Automation*, pages 955–958. IEEE, 2015.

[25] Luis Eduardo Jaramillo. Malware threats analysis and mitigation techniques for compromised systems. In *Journal of Information Systems Engineering & Management*, volume 4(1). Modestum, 2019.

[26] Carlos Jensen, Scott King, and Victor Kuechler. Joining free/open source software communities: An analysis of newbies' first interactions on project mailing lists. In *System Sciences (HICSS), 2011 44th Hawaii International Conference on*, pages 1–10. IEEE, 2011.

[27] M Khan and Faizan UrRehman. Free and open source software: Evolution, benefits and characteristics. *International Journal of Emerging Trends & technology in Computer Science*, 1(3):1–7, 2012.

[28] Karim R Lakhani and Eric Von Hippel. How open source software works:free? user-to-user assistance. In *Produktentwicklung mit virtuellen Communities*, pages 303–339. Springer, 2004.

[29] Karim R Lakhani and Robert G Wolf. Why hackers do what they do: Understanding motivation and effort in free/open source software projects. 2003.

[30] Shyong K Lam and John Riedl. Shilling recommender systems for fun and profit. In *WWW*, pages 393–402. ACM, 2004.

[31] Blerina Lika, Kostas Kolomvatsos, and Stathes Hadjiefthymiades. Facing the cold start problem in recommender systems. *Expert Systems with Applications*, 41(4):2065–2073, 2014.

[32] Chao Luo, Hiroyuki Okamura, and Tadashi Dohi. Optimal planning for open source software updates. *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability*, 230(1):44–53, 2016.

[33] Martin Michlmayr. Quality improvement in volunteer free software projects: Exploring the impact of release management. In *Proceedings of the First International Conference on Open Source Systems*, pages 309–310, 2005.

[34] Martin Michlmayr, Francis Hunt, and David Probert. Release management in free software projects: Practices and problems. In *IFIP International Conference on Open Source Systems*, pages 295–300. Springer, 2007.

[35] Audris Mockus, Roy T Fielding, and James Herbsleb. A case study of open source software development: the apache server. In *Proceedings of the 22nd international conference on Software engineering*, pages 263–272, 2000.

[36] Ranjit Nayak, Sridhar Sudarsan, Vishwanath Venkataramappa, Qinhua Wang, and Leigh Williamson. Automated deployment of an application, December 29 2005. US Patent App. 10/874,495.

[37] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. CHAINIAC: Proactive software-update transparency via collectively signed skipchains and verified builds. In *26th USENIX Security Symposium 17*, pages 1271–1287, 2017.

[38] Michael P O'Mahony, Neil J Hurley, and Guénolé CM Silvestre. Recommender systems: Attack types and strategies. In *AAAI*, pages 334–339, 2005.

[39] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. *ACM SIGCOMM Computer Communication Review*, 42(4):323–334, 2012.

[40] Paul Resnick and Hal R Varian. Recommender systems. *Communications of the ACM*, 40(3):56–58, 1997.

[41] Francesco Ricci, Lior Rokach, and Bracha Shapira. Introduction to recommender systems handbook. In *Recommender systems handbook*, pages 1–35. Springer, 2011.

[42] Malek Ben Salem, Shlomo Hershkop, and Salvatore J Stolfo. A survey of insider attack detection research. In *Insider Attack and Cyber Security*, pages 69–90. Springer, 2008.

[43] Julien Savaux, Julien Vion, Sylvain Piechowiak, René Mandiau, Toshihiro Matsui, Katsutoshi Hirayama, Makoto Yokoo, Shakre Elmane, and Marius Silaghi. Discsps with privacy recast as planning problems for self-interested agents. In *2016 IEEE/WIC/ACM International Conference on Web Intelligence (WI)*, pages 359–366. IEEE, 2016.

[44] Julien Savaux, Julien Vion, Sylvain Piechowiak, René Mandiau, Toshihiro Matsui, Katsutoshi Hirayama, Makoto Yokoo, Shakre Elmane, and Marius Silaghi. Utilitarian approach to privacy in distributed constraint optimization problems. In *The Thirtieth International Flairs Conference*, 2017.

[45] Julien Savaux, Julien Vion, Sylvain Piechowiak, René Mandiau, Toshihiro Matsui, Katsutoshi Hirayama, Makoto Yokoo, Shakre Elmane, and Marius Silaghi. Stochastic game modelling for distributed constraint reasoning with privacy. In *ISAIM*, 2018.

[46] Julien Savaux, Julien Vion, Sylvain Piechowiak, René Mandiau, Toshihiro Matsui, Katsutoshi Hirayama, Makoto Yokoo, Shakre Elmane, and Marius Silaghi. Privacy stochastic games in distributed constraint reasoning. *Annals of Mathematics and Artificial Intelligence*, pages 1–25, 2019.

[47] Julien Savaux, Julien Vion, Sylvain Piechowiak, René Mandiau, Toshihiro Matsui, Katsutoshi Hirayama, Makoto Yokoo, and Marius Silaghi. Privacité dans les discsp pour agents utilitaires. In *JFSMA*, pages 129–138, 2016.

[48] Walt Scacchi. Free/open source software development. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 459–468, 2007.

[49] Walt Scacchi. Free/open source software development. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 459–468. ACM, 2007.

[50] Walt Scacchi. Collaboration practices and affordances in free/open source software development. In *Collaborative software engineering*, pages 307–327. Springer, 2010.

[51] Walt Scacchi, Joseph Feller, Brian Fitzgerald, Scott Hissam, and Karim Lakhani. Understanding free/open source software development processes. *Software Process: Improvement and Practice*, 11(2):95–105, 2006.

[52] J Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. Collaborative filtering recommender systems. In *The adaptive web*, pages 291–324. Springer, 2007.

[53] J Ben Schafer, Joseph A Konstan, and John Riedl. Meta-recommendation systems: user-controlled integration of diverse recommendations. In *11th international conference on Information and knowledge management*, pages 43–51. ACM, 2002.

[54] M Silaghi, Song Qin, Khalid Alhamed, Toshihiro Matsui, Makoto Yokoo, and Katsutoshi Hirayama. P2P petition drives and deliberation of shareholders. In *Decentralized Coordination*, 2013.

[55] Marius C Silaghi, Khalid Alhamed, Osamah Dhannoon, Song Qin, Rahul Vishen, Ryan Knowles, Ihsan Hussien, Yi Yang, Toshihiro Matsui, Makoto Yokoo, et al. DirectDemocracyP2P – decentralized deliberative petition drives –. In *P2P*, 2013.

[56] Marius-Calin Silaghi and Boi Faltings. A comparison of distributed constraint satisfaction techniques with respect to privacy. In *Proceedings of the Distributed Constraint Reasoning Workshop at AAMAS*, 2002.

[57] Marius-Calin Silaghi and Boi Faltings. Openness in asynchronous constraint satisfaction algorithms. In *3rd DCR-02 Workshop*, pages 156–166, 2002.

[58] Marius-Calin Silaghi and Gerhard Friedrich. Secure stochastic multi-party computation for combinatorial problems and a privacy concept that explicitely factors out knowledge about the protocol. *IACR Cryptology ePrint Archive*, 2005:154, 2005.

[59] Jose M Such, Agustín Espinosa, and Ana García-Fornes. A survey of privacy in multi-agent systems. *The Knowledge Engineering Review*, 29(3):314–344, 2014.

[60] Jonathan Turpie. Integrating software updates with the testing and deployment of software, May 9 2017. US Patent 9,645,808.

[61] Kim O Van Camp, Bob Wiggins, and Naqi Syed. Automated deployment of computer-specific software updates, April 7 2015. US Patent 9,003,387.

[62] Greg R Vetter. Commercial free and open source software: knowledge production, hybrid appropriability, and patents. *Fordham Law Review*, 77(2087), 2009.

[63] Georg Von Krogh, Stefan Haefliger, Sebastian Spaeth, and Martin W Wallin. Carrots and rainbows: Motivation and social practice in open source software development. *Mis Quarterly*, 36(2):649–676, 2012.

[64] Sam Williams. *Free as in Freedom*. O'Reilly Media, 2002.

[65] Fuguo Zhang. Average shilling attack against trust-based recommender systems. In *2009 International Conference on Information Management, Innovation Management and Industrial Engineering*, volume 4, pages 588–591. IEEE, 2009.