Framework for Open-Source Software Update Mechanism: Resistance to Stacking the Deck Attack

by

Khalid Alhamed

A dissertation submitted to
Florida Institute of Technology
in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Melbourne, Florida
May 2015
We the undersigned committee hereby approve the attached dissertation

Framework for Open-Source Software Update Mechanism: Resistance to Stacking the Deck Attack

by
Khalid Alhamed

__________________________
Marius Silaghi, Ph.D.
Associate Professor, Computer Sciences and Cybersecurity
Committee Chair, Thesis Advisor

__________________________
William Allen, Ph.D.
Associate Professor, Computer Sciences and Cybersecurity
Thesis Committee

__________________________
Phil Bernhard, Ph.D.
Associate Professor, Computer Sciences and Cybersecurity
Thesis Committee

__________________________
Muzaffar Shaikh, Ph.D.
Professor, Engineering Systems
Thesis Committee

__________________________
Richard Ford, Ph.D.
Professor, Computer Sciences and Cybersecurity
Department Head
Abstract

Title: Framework for Open-Source Software Update Mechanism: Resistance to Stacking the Deck Attack
Author: Khalid Alhamed
Committee Chair: Marius Silaghi, Ph.D.

Currently many users trust open-source binaries downloaded from repositories such as sourceforge.net, github.com, and gitorious.org. As with any system connected to the Internet, such repositories can be subject to attacks tampering with the distributed binaries (inserting malicious code, changing behavior). Developers can change their vision and abandon features that are essential for certain users. Moreover, well-funded attackers can effectively take control of a project by orchestrating the transfer of the leadership of the developers to people whom they control. We propose a framework to reduce the level of trust that users are required to have into updates for open-source software that is maintained by volunteers. This framework integrates evaluations from independent testers into the mechanism for automatic updates of binaries for free and open-source software. Each user can select a set of testers he or she trusts and can limit automatic updates to the case where a certain quality is evaluated by these testers with a minimum declared depth of test, when aggregated
with a configurable function. In fact, with the proposed method, it is sufficient for the user to trust that his flexibly-specified constellation of independent testers is safe to each given attack, even as all may be subject to different attacks. Our solution is adapted to the peer-to-peer (P2P) environment, without centralized control, to enhance the independence of the testers. In such environments, each peer is equipped with a distributed recommender agent to propagate recommendations about the peer’s trusted testers. The proposed distributed recommender model enforces the independence of the testers by: 1) Automatic amortization of the recommendations when the trust is not manually strengthened, and 2) Blocking propagation at any user that does not employ the reviews. In turn, this independence of the testers is used to enhance the security of the automatic updating system. A new integrated framework of open-source development, testing, distribution, and updating is defined, implemented, and made available.
# Contents

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xi</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Software Update Stages

1.1.1 Maintenance Stage

1.1.2 Distribution Stage

1.1.3 Downloading and Installing Stage

1.2 Update Mechanism for Open-Source Software

1.2.1 Overview of OSS Development and Update Model

1.2.2 Trust Issues in Updating Open-Source Software

1.3 Motivation

1.4 Contributions

1.5 Overview of the Chapters

## 2 Literature Review

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
</tr>
<tr>
<td>Section</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>2.1 Software Updating</td>
</tr>
<tr>
<td>2.1.1 Reliability</td>
</tr>
<tr>
<td>2.1.2 Security</td>
</tr>
<tr>
<td>2.1.3 Availability</td>
</tr>
<tr>
<td>2.1.4 Usability</td>
</tr>
<tr>
<td>2.1.5 Efficiency</td>
</tr>
<tr>
<td>2.2 Updating Open-Source Software</td>
</tr>
<tr>
<td>2.3 Recommender Systems</td>
</tr>
<tr>
<td>2.3.1 Content-Based Recommendations (Content-Based Filtering)</td>
</tr>
<tr>
<td>2.3.2 Collaborative Recommendations (Collaborative Filtering)</td>
</tr>
<tr>
<td>2.3.3 Hybrid Approaches</td>
</tr>
<tr>
<td>2.3.4 Architecture-based Classifications</td>
</tr>
<tr>
<td>2.4 Free Riding in P2P Systems</td>
</tr>
<tr>
<td>3 Secure Auto-Update for Open-Source Software</td>
</tr>
<tr>
<td>3.1 Introduction</td>
</tr>
<tr>
<td>3.2 Contributions</td>
</tr>
<tr>
<td>3.3 Threat Model</td>
</tr>
<tr>
<td>3.4 Concepts</td>
</tr>
<tr>
<td>3.4.1 Background Concepts</td>
</tr>
<tr>
<td>3.4.2 Introduced Concepts</td>
</tr>
<tr>
<td>3.5 Architecture</td>
</tr>
<tr>
<td>3.6 Decision Making for Accepting Automatic Updates</td>
</tr>
<tr>
<td>3.7 Updates Protocol</td>
</tr>
<tr>
<td>3.8 Data Formats for Information Exchange</td>
</tr>
</tbody>
</table>
# Distributed Recommendation of Testers

## 4

### 4.1 Introduction

#### 4.1.1 Reference Implementation

### 4.2 Motivation: Desirable Properties

### 4.3 Recommender System Problem

#### 4.3.1 Difference from the Classic Recommender System Problem

### 4.4 The Mechanism of Recommending Testers

#### 4.4.1 Formalizing Heuristic-based Recommendations

#### 4.4.2 The Frequency of Tester Replacement

#### 4.4.3 Distributed Recommender Model in Each Peer

#### 4.4.4 Tester's Life-Cycle

### 4.5 Evaluation And Experiments

#### 4.5.1 Evaluation Strategies

#### 4.5.2 Experiments

#### 4.5.3 Evaluation of Distribution

#### 4.5.4 Global Stability

#### 4.5.5 Casualties

### 4.6 Conclusions

# Supernode Services based on Incentives

## 5

### 5.1 Motivation

### 5.2 Introduction

### 5.3 System Architecture

#### 5.3.1 Identity Management
5.3.2 Incentive Management ........................................ 115
5.3.3 Supernode Management .................................... 116
5.3.4 Communication Management ............................... 117
5.4 Protocols .......................................................... 119
5.5 Methodology and Experiments ................................. 128
  5.5.1 Why Simulations and Survey? ............................ 129
  5.5.2 Survey Design and Implementation: ..................... 129
  5.5.3 Survey Statistics .............................................. 131
  5.5.4 System Simulation ............................................ 133
5.6 Conclusions ....................................................... 141

6 Conclusions and Discussions .................................... 144
  6.1 Summary .......................................................... 144
  6.2 Limitations and Future Work ................................. 146

Bibliography .................................................................. 148

A DirectDemocracyP2P System .................................... 170

B Online Survey for Incentive-based P2P Chat System .... 173

C List of Publications .................................................. 178
# List of Figures

1.1 Software Update Stages ................................................. 3  
1.2 A typical Software Update System ......................................... 5  
1.3 Open-Source Development and Update Model .............................. 8  

2.1 Software Update Taxonomy ................................................ 16  
2.2 Cluster Computer System .................................................. 17  
2.3 Safe Load ................................................................. 19  
2.4 Metadata for Adobe Flash Player .......................................... 22  
2.5 *Partial Upgrade* Offered When Trying to Update Ubuntu ............ 34  

3.1 Overall Architecture of Integrated Development, Testing and Updates . 59  
3.2 Update Request ........................................................... 68  
3.3 Update Response ............................................................ 69  
3.4 Update Descriptor .......................................................... 70  

4.1 Dynamic Scoring Matrix for Eye Color and Shirt Color .................. 79  
4.2 Exchanging Testers Information in the Proposed Peer-to-Peer Recommender System ............................................. 80  
4.3 Model I: Overall Architecture of the Recommender System ............ 84
<table>
<thead>
<tr>
<th>Section Number</th>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4</td>
<td>Model II: Overall Architecture of the Recommender System</td>
<td>87</td>
</tr>
<tr>
<td>4.5</td>
<td>Distributed Recommender Model in Each Peer with Example</td>
<td>91</td>
</tr>
<tr>
<td>4.6</td>
<td>Screenshots of the Integrated Recommender System in DDP2P</td>
<td>98</td>
</tr>
<tr>
<td>4.7</td>
<td>Number of Users Trusting Each Tester After 100 Rounds</td>
<td>106</td>
</tr>
<tr>
<td>4.8</td>
<td>Global Stability</td>
<td>107</td>
</tr>
<tr>
<td>4.9</td>
<td>Casualties after 50 rounds</td>
<td>108</td>
</tr>
<tr>
<td>5.1</td>
<td>Providing Incentives for Relaying Messages</td>
<td>113</td>
</tr>
<tr>
<td>5.2</td>
<td>P2P System Architecture</td>
<td>114</td>
</tr>
<tr>
<td>5.3</td>
<td>Terms Agreement</td>
<td>116</td>
</tr>
<tr>
<td>5.4</td>
<td>Subscribing With One or More Directories</td>
<td>117</td>
</tr>
<tr>
<td>5.5</td>
<td>Case 1: Roaming Peer (No NAT)</td>
<td>123</td>
</tr>
<tr>
<td>5.6</td>
<td>Case 2: Full-Cone NATs</td>
<td>124</td>
</tr>
<tr>
<td>5.7</td>
<td>Case 3: Symmetric NAT</td>
<td>125</td>
</tr>
<tr>
<td>5.8</td>
<td>Case 4: NAT at Destination</td>
<td>126</td>
</tr>
<tr>
<td>5.9</td>
<td>Reducing latency</td>
<td>127</td>
</tr>
<tr>
<td>5.10</td>
<td>Integrating Survey and Simulations</td>
<td>131</td>
</tr>
<tr>
<td>5.11</td>
<td>Comparing Incentives</td>
<td>133</td>
</tr>
<tr>
<td>5.12</td>
<td>SN Availability During the Day</td>
<td>136</td>
</tr>
<tr>
<td>5.13</td>
<td>Bayesian Network</td>
<td>137</td>
</tr>
<tr>
<td>5.14</td>
<td>Only use ”Revealing Cause” as an Incentive</td>
<td>141</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Software Update Systems and the Authenticated Information They Provide ............. 24
2.2 Software Update Common Attacks ................................................................. 29
2.3 A 4 × 4 Rating Matrix for a Recommender System ........................................ 40
3.1 QD and Testers certificates on QoT & RoT .................................................... 58
5.1 Survey Statistics ......................................................................................... 132
5.2 Peer Instance Properties ........................................................................... 134
5.3 CPT Table ................................................................................................. 138
5.4 Unique Incentive ....................................................................................... 139
5.5 Absence of Incentives: Not Accounting for Free Services ......................... 140
## List of Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma$</td>
<td>release sources</td>
</tr>
<tr>
<td>$\nu$</td>
<td>version identifier (i.e., 1.2.0)</td>
</tr>
<tr>
<td>$\Phi$</td>
<td>quality definitions added by a developer</td>
</tr>
<tr>
<td>$d$</td>
<td>release date</td>
</tr>
<tr>
<td>$\tau$</td>
<td>tester ID</td>
</tr>
<tr>
<td>$\beta$</td>
<td>binary software</td>
</tr>
<tr>
<td>$\eta$</td>
<td>information for release files</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>release building parameters</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>Boolean flag, false for $\varepsilon = \bot$</td>
</tr>
<tr>
<td>$t$</td>
<td>the date of the test data</td>
</tr>
<tr>
<td>$\Upsilon$</td>
<td>quality definitions added by a tester</td>
</tr>
<tr>
<td>$\Theta$</td>
<td>vector of Qualities of Tests</td>
</tr>
<tr>
<td>$\Psi$</td>
<td>vector of the Result of Tests</td>
</tr>
<tr>
<td>$A$</td>
<td>tester address</td>
</tr>
<tr>
<td>$\mathcal{W}$</td>
<td>tester weight</td>
</tr>
<tr>
<td>$\mathcal{P}$</td>
<td>message sender</td>
</tr>
<tr>
<td>$S$</td>
<td>digital signature</td>
</tr>
<tr>
<td>$\delta$</td>
<td>secret key</td>
</tr>
<tr>
<td>$\bot$</td>
<td>empty value (i.e. null)</td>
</tr>
</tbody>
</table>
Acknowledgements

I owe a tremendous debt of gratitude to all who helped to make this dissertation possible. My thanks go first and foremost to my mother and my father who have always been understanding and supportive. Equally passionate thanks go to my wife who has granted me a great amount of love and patience and accompanied me through many working days, nights, and weekends.

I also extend my deepest gratitude to my friend and advisor, Dr. Marius Silaghi. His guidance and support were invaluable in the course of this work. I also thank him for extending our relationship beyond the lab into families.

I want to thank my committee members, Dr. William Allen, Dr. Phil Bernhard, and Dr. Muzaffar Shaikh. Their comments, understanding, and interest in my work were extremely encouraging.

I am also thankful to my friends and colleagues, Sultan Almutairi and Yasser Howsawi for their support and help during my work.

I owe many thanks to the Saudi Arabia government for supporting my study and funding my trips to several conferences.
Declaration

I declare that the work in this thesis is solely my own except where attributed and cited to another author. Most of the material in this thesis has been previously published by the author. For a complete list of publications, please refer to Appendix C at the end of this thesis.
Chapter 1

Introduction

Actively used software systems tend to continually evolve to ensure their utility and safety [13, 76, 16]. Software evolution demands more effort from vendors to keep the software stable and reliable by fixing errors, enhancing performance, adding new features, removing obsolete features, solving security issues, and keeping up with evolving requirements. On the other hand, it is important for users to keep their software up to date to take advantage of the new enhancements and features. Such mutual demands between users and vendors lead to implementing various methods and technologies for the software updating process.

An essential feature of any software application is the provision of an automatic update system which provides a convenient way to fix and enhance the software application. The automatic update of software systems include discovering, downloading, and installing the latest versions of software without needing manual intervention by the user [128]. Without automatic updates, users are exposed to new and unexpected security attacks. Lack of automatic updates also leads to long term coexistence of many versions, thus increasing the complexity of ensuring interoperation.
Automatic updating of Free and Open-Source Software (FOSS) is a special case of software update, more complex when compared with proprietary software (close-source). The issue of trust, between an open-source project and its users, is a major challenge for update propagation in the current open-source model [106].

Our research focuses on reducing the level of trust that users are required to have in the developers and the distributors of free and open-source software.

We propose an integrated framework for development, testing and secure automatic updates distribution of free and open-source software. The framework yields an architecture for automatic updates based on signed recommendations from a user-defined constellation (e.g., fraction) out of a set of testers that the user trusts. Such trust in testers can be gained based on a distributed recommender system and prior experiences with the testers.

In this chapter, we first give an overview of software update stages and then explore issues in the typical OSS development and updating process. Finally, we discuss the motivation behind our research and the offered contributions.

1.1 Software Update Stages

The software update process goes in three main stages: Maintenance Stage, Distribution Stage, and Downloading & Installing Stage as shown in Figure 1.1.
1.1.1 Maintenance Stage

Software vendors are required to keep their software stable and reliable by fixing errors, enhancing performance, adding new features, removing obsolete features, solving security issues, and keeping up with evolving requirements. A common perception of software maintenance is that it is only about fixing bugs. In fact, bug-fixing represents only approximately 21% of the software maintenance effort [41], while, over 80% of the maintenance effort is focused on non-corrective actions [115] (e.g., applying new requirements, adding new features, removing obsolete features, and solving security issues).

1.1.2 Distribution Stage

In this stage, the concern is how to inform clients about the new updates and how to supply the new updates to them. Software updates distribution can take several forms: using a centralized server to update the connected clients as done in enterprises and companies. An example is "Windows Server Update Services" (WSUS) that enables
administrators to auto-manage the new updates for Microsoft products in a corporation. WSUS can be set to download new updates from the Microsoft Update website and then propagate them to all or specific computers on a network. Other distribution mechanisms depend on preinstalled package managers, application updaters, or library managers on clients machines that keep pulling the new updates from software update mirrors (servers that host the update files). These mirrors can be owned by the software vendors (private mirrors) or can be public mirrors. Clients can reach mirrors through various network techniques (Internet, P2P, OTA, or wireless). Alternatively, software vendors can provide new software updates to their clients directly via email, by mail (store updates on storage media such as DVD/CD, flash, or others), or through their websites on the Internet (e.g., discussion groups or forums).

1.1.3 Downloading and Installing Stage

In this stage, clients can use preinstalled package managers, application updaters or library managers to download the desired updates and then install them on their machines. Security flags can be raised in this stage (authentication, integrity,..). Reliability issues are also important during the installation stage. Figure 1.2 shows each software update stage interaction in a typical software update system.
1.2 Update Mechanism for Open-Source Software

Open-Source Software (OSS) is not only about allowing users to access the source-code of the software. “Open source is a term ... to describe the tradition of open standards, shared source code, and collaborative development behind software such as the Linux and FreeBSD operating systems” [105]. Open-source software can be classified based on distribution terms and the type of the software license. In this study, we focus on Free and Open-Source Software (FOSS) [42]. FOSS is governed by licenses that permit users to freely use, study, change, and redistribute the software (e.g., OSS with GPL license). This kind of license encourages users to voluntarily cooperate and improve the software. Such cooperation between users and open-source projects leads to decreased software costs, increased security and stability, improved privacy, and it gives users more control over the software. This is different from proprietary software (“closed software”), where the software is restricted with copyright, and most of the time the source-code is hidden from the users. There has been much successful open-source software such as the Linux
operating system, the internet browser Mozilla Firefox, the Apache HTTP server, and the Python language. [105].

1.2.1 Overview of OSS Development and Update Model

In order to detail the security issues of the current update systems for OSS, we should first clarify how the development and update process works in OSS projects. Figure 1.3 shows a typical development and update model for an OSS project [153], which has six components:

1. **Trusted Repository (main repository)**: It is usually a public access storage (e.g., a web server) where people can get the new releases of an official version of the software. It can be also a resource for other related information like mailing lists, documentation, and bug tracking systems.

2. **Trusted Developer**: Only the trusted developers have the permission to modify the trusted repository directly. In the beginning stage of a project, the trusted repository is created by a small number of developers who we refer to as trusted developers. They have the right to allow any other developer to become a trusted developer and to have access to the trusted repository.

3. **Developer**: The other developers (non-trusted developers) can download copies of the software from the trusted repository and make changes to their local copies. However, they are required to submit their changes to a trusted developer who can upload other developers’ changes into the trusted repository.

4. **Distributor**: Distributors download the source-code of the new releases from the trusted repository and produce the binaries of the software by building and
compiling the source-code. They may provide additional value to the software (e.g., testing, adding components, or support). Distributors report the software bugs to the project reporting repository, which is usually a part of the trusted repository to be considered in the following releases.

5. **Mirror:** Distributors post the new releases on their own repositories (private mirrors) or use third party repositories (public mirrors). It is common that OSS uses mirrors to distribute the software. Mirrors provide fault tolerance (if a mirror is down then users can employ other mirrors) and offload traffic (bandwidth costs) from the distribution’s main repository. In general, open-source communities depend on donated public mirrors to distribute their software and updates.

6. **User:** Users have the choice of either downloading the software directly from the trusted repository or downloading it from their trusted distributors. They report bugs either to their trusted distributors or to the trusted repository. There are two types of users: end-users, who use the software without changing the code, and *users as developers* [153], who may change the code and submit their changes to the trusted developers.
1.2.2 Trust Issues in Updating Open-Source Software

The idea of this research is to identify and solve issues regarding automatic updating of FOSS. We summarized these issues as the following:

- **Trust issues during the development stage:** Open-source software (OSS) is developed by volunteers with different incentives (reputation, solving problems, interest of research,...). Volunteer programmers spend most of the time on the development of the software [101] (where the challenge is!) but they give less effort in software testing (considered less challenging). This fact affects the reliability of the software updating. Besides that, developers can change their vision and abandon features that are essential for certain users. Moreover, well
funded attackers can effectively take control of a project by orchestrating the
transfer of the leadership of the developers to people that they control.

- **Trust issues during the distribution stage:** OSS distribution depends on
donated public mirrors (third-party servers) to distribute the software to end-users.
Software update systems should be able to recognize malicious mirrors and block
out updates from them [23, 65].

- **Trust issues during the building and compiling stage:** Regarding the fact
that the source-code of an OSS can be recompiled and redistributed under
different branch names (e.g., OSS with GPL license), the update process is more
complex when compared to proprietary software (e.g., deciding which branch is
trustworthy to be used for software update).

### 1.3 Motivation

When it comes to mergers and acquisitions in the software industry, end-users are always
concerned about the impact of the acquisition on their privacy and security (e.g., Skype
acquisition in 2011 [88] and WhatsApp acquisition in 2014 [29]. The question is: "If
these systems are free and open-source systems, would they have the same problem?"
Can someone take over FOSS systems and control their development and distribution?
Let’s consider the following scenario: Assume a free GPL open source agent system for
supporting petition drives, like DirectDemocracyP2P (see Appendix A for more details).
For the success of such a system, supposed to be under heavy development even after
its release, it is essential to have an automated update mechanism.
Given the main application of such a system, namely petition drives, let us assume that eventually somebody will use it to get signatures for a petition that is not liked by BIG MONOPOLY INC. Currently, one of the easiest ways for BIG MONOPOLY INC. to fight against the petition drive process is to disrupt the DDP2P software via its automatic updates. Since DDP2P is maintained by volunteers, the attack would proceed as follows.

First, BIG MONOPOLY INC. officers can offer a good job to the volunteer maintainers and can sponsor people that they control to take over the development. Once BIG MONOPOLY INC controls the DDP2P maintenance process, they can introduce features that disable or somehow disrupt the petition drive processes annoying them. These attacks can be so surreptitious that it may take a long time for any user to even realize what is happening. For example, an automatic update can selectively slow down the dissemination of the votes for one targeted petition. By the time users detect the attack, the whole agent system may be compromised and its data may be irremediably lost. Unfortunately, even volunteers observing it lack a mechanism to warn other users. Let us refer to this attack as the *Stacking the Deck Attack*.

### 1.4 Contributions

This thesis presents different but related contributions towards building a secure update framework for FOSS. First we address the problem of a potential attack in FOSS, Stacking the Deck Attack (SDA). Then we introduce a solution based on integrating evaluations from independent testers into the mechanism of automatic updating of FOSS. Another contribution is a support mechanism to enhance testers independence by
Propagating recommendations along direct connections in a given peer-to-peer network.

The contributions of the thesis are as follows:

- **Defining Stacking the Deck Attack (SDA) in the process of updating FOSS.**
  
  We introduce a new attack, Stacking the Deck Attack (SDA), that can occur during the process of updating free and open-source software. SDA can be classified as a *social engineering* attack that employs *psychological manipulation* of OSS developers by an attacker (e.g., external beneficiary) to perform certain actions for the purpose of taking control of the OSS development and distribution. According to [Braiker 2004], there are several mechanisms that manipulators can use to control their victims with positive reinforcement including money, gifts and public recognition.

- **A solution for SDA attack.** We propose a solution for mitigating the SDA attack by inserting independent intermediaries (testers) between the developers and the end-users, intermediaries that can eventually detect and easily warn or hamper the automatic damage of the whole system. Once a Stacking the Deck Attack is detected, other volunteers can start or recommend new branches (given that the software is free and open-source).

- **Designing and evaluating a recommender system to meet the principles of the proposed framework.** A design of a recommendation system is proposed to recommend testers to users via a distributed P2P agent network. The recommendation system is designed to meet the principles of the framework:

  1. The recommendation procedure should not be under the control of a limited number of users.
2. Proximity: giving priority to testers that are close to (or far from) the user.

3. Diversity: giving priority to testers that are used by fewer neighbors, in terms of some social network.

- **An incentive-based P2P System for handling connection services.** In order to distribute recommendations for testers, we rely on a P2P network to exchange testers’ recommendations. We address the free-riding problem of the P2P network. We propose a solution based on formalizing and identifying supernode incentives that can reduce the free-riding problem. We have designed a protocol that embeds incentives in the STUN protocol.

### 1.5 Overview of the Chapters

The remainder of this thesis is organized into the following chapters:

Chapter 2 provides a literature review of related topics and studies. The literature review focus on three areas relevant to this thesis. First, we present an overview of the existing methods and applications of a software update. Secondly, we investigate various models that are used for distributed recommender systems. Finally, we introduce the problem of free-riding in P2P systems regarding the communications between peers and review state-of-the-art solutions.

In Chapter 3, we examine the situation that currently, many users trust binaries downloaded from repositories such as sourceforge.net. As with any system connected to the Internet, such repositories can be subject to attacks tampering with the distributed binaries (inserting viruses, changing behavior). We present a framework for integrating evaluations from independent testers into the mechanism of automatic
updating of binaries for FOSS developed by volunteers. This framework is defined, implemented and made available in order to limit the risk of a new attack, Stacking the Deck Attack (SDA), that can exploit the process of updating FOSS.

Chapter 4, discusses the essential role that testers play in our mechanism for auto-updating of FOSS. Their independence is essential for the resistance to Stacking the Deck Attacks (where the attacker can orchestrate to take over the control of the test process). To encourage independence of the testers, essential for the desired security, a distributed P2P recommendation mechanism is employed, suggesting testers for end-users based on preferences of immediate connections and on the frequency of usage of these testers in the neighborhood of the user. Metrics of success and experiments for identifying promising parameters are reported in this chapter.

In Chapter 5, we introduce a new fully decentralized unstructured peer-to-peer (P2P) approach to open-source instant messaging systems. In this approach each human owning a peer can control the traffic supported by her system. The control may be based on criteria such as: (a) her desire to help the endpoints of the communication based on friendship, (b) her desire to help a cause based on the content/topic of the communication, (c) reputation and rewards, or (d) her interest in the handled data itself. In P2P environments, providing intrinsic incentives for peers to help with traffic is important for an open-source freeware chat system to eventually be viable. In non-incentive P2P systems like Skype, the availability of open-source versions can potentially starve the system of supernodes (once users learn how to disable the resource consuming supernode function).

In Chapter 6, we summarize the contributions of this thesis, discuss limitations of the work, and highlight the important directions of future work.
Chapter 2

Literature Review

In this chapter, we provide a literature review of topics and studies relevant to this research. In general, the literature on software updates is broad and wide ranging, with numerous existing methods and applications, as we can see in Section 2.1. In Section 2.2, we review the challenges and solutions to updating open-source software. Decisions on adopting updates can be improved by using recommendations. Our problem is identified in a peer-to-peer (P2P) system. In Section 2.3, we introduce an overview of the field of recommender systems and discuss the current classifications of recommendation methods. In Section 2.4, we provide a review of free riding problems and solutions in P2P networks.

2.1 Software Updating

Software updating is the process of transforming software from its current state to a new updated state. Updating software is typically an arduous, error prone, and risky task since a botched installation can prevent further fixes and updates.
Automatic software update systems are used to discover, download, and install the new updates of software without needing manual intervention by the user [128]. Each software update system has certain properties and characteristics based on the nature (purpose) of the software and the deployment environment. For example, some software update systems offer high availability by applying updates live, online and with no downtime, e.g., updates for real time systems [151, 152, 133]. Some other updates are designed to offer high performance updating by dividing the updates into small chunks (update packages) for faster I/O over the networks and for supporting partial (accumulate) updating, e.g., the Partial Upgrade mode in Update Manager for Ubuntu [27].

Sometimes the software handles its own updates so that the software can update itself with no need for manual intervention by the user or external software, such as the Firefox browser [97], while in other cases the operating system performs the automatic updates via package managers such as YUM, RPM, Slaktool, and ports for BSD systems [23, 50]. Library managers are another common way of handling updates for programming language environments to install and update optional libraries such as Python’s easy install [147].

Significant work exists on updates of software. Based on the recent research results on updating software, we can roughly categorize software update properties and characteristics as update reliability, update security, update availability, update performance, and update usability (see Figure 2.1). In this section, a short discussion is given for each property to identify research directions and problems that deserve further investigation.
2.1.1 Reliability

"Reliability is one area where continuous improvement is desired" [40]. In this direction, it has been investigated how to move (update) a software from the current state to another enhanced reliable state. Many researchers are focusing on enhancing software updating reliability by introducing new methods and solutions to ensure correctness and robustness of the updates. The idea here is to find robust solutions for software updating problems that cause bugs, errors and unexpected behaviors. In software engineering, this kind of problem which affects software reliability after updating a system is called "software regression" [103].

Research in update reliability can be divided into:

- Fault tolerance/running update with recovery

<table>
<thead>
<tr>
<th>Reliability</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Post-update actions</td>
<td>Analyzing users feedback after updating.</td>
</tr>
<tr>
<td>Pre-update actions</td>
<td>Preventing failure state by testing the updates before using them.</td>
</tr>
<tr>
<td>Security</td>
<td>Without authentication, software update can be exploited by malicious programs.</td>
</tr>
<tr>
<td>Attacks</td>
<td>Preventing software update common attacks.</td>
</tr>
<tr>
<td>Security in updating OSS</td>
<td>Solving security issues raised in the process of updating open-source software.</td>
</tr>
<tr>
<td>Availability</td>
<td>Meeting special constraints while updating programs in the execution time.</td>
</tr>
<tr>
<td>Real-time and embedded systems</td>
<td>Identifying advantages and disadvantages of current DSU systems in programming lang</td>
</tr>
<tr>
<td>Database Systems</td>
<td>Solutions for schema evolution in embedded database systems for DSU.</td>
</tr>
<tr>
<td>Operating Systems</td>
<td>Overview on the most recent DSU systems in OS.</td>
</tr>
<tr>
<td>Usability</td>
<td>Enhancing the ease of use and flexibility in updating systems.</td>
</tr>
<tr>
<td>Ease of use and flexibility</td>
<td>Methods for reducing requirements for applying DSU.</td>
</tr>
<tr>
<td>Usability issues in DSU</td>
<td>Identifying the effects of the development mechanisms in OSS on the software usability.</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Identifying efficiency measurements factors in Real-time and Embedded Systems.</td>
</tr>
<tr>
<td>Efficiency in DSU Systems</td>
<td>Identifying efficiency measurements factors in DSU systems.</td>
</tr>
<tr>
<td>Efficiency and update security</td>
<td>Methods for increasing efficiency in update authentication.</td>
</tr>
<tr>
<td>Software updates distribution</td>
<td>Increasing the efficiency regarding to software updates distribution.</td>
</tr>
</tbody>
</table>

Figure 2.1: Software Update Taxonomy
• Analyzing user feedback after updating/postactive actions/post-update actions

• Preventing failure state/testing before using/proactive actions/pre-update actions.

Fault tolerance/running update with recovery

To avoid the adverse impacts of unsuccessful upgrades, researchers have investigated fault-tolerance system upgrade methods [143]. Several methods have been implemented for software update recovery. Some methods enhance software update recovery in a "clustered computer system" as in enterprise systems and e-commerce systems [40]. Figure 2.2 shows a solution proposed by Roy P. D'Souza [40] for updating a clustered computer system. The routing mechanism has been enhanced with the ability of studying the capability of each server in the cluster (software modules versions, performance) for more reliably redirecting requests. Such routers and servers are called intelligent director agents (IDA). This could help updating "clustered computer systems" in parallel.

Figure 2.2: Cluster Computer System
Other research tries to apply parallel updating in non-cluster environments. A recent suggestion for addressing unstable updates is to have users install the new versions in parallel with old versions, in a so-called multi-version software update approach [22, 61]. This approach introduces strong requirements on the whole code development (where all different versions have to be designed to coexist and share data), and the provided prototype requires the capture of all system calls.

Some other research focuses on software update recovery in embedded systems (firmware). Even though firmware is not intended to be changed, updates in firmware are frequently needed in order to correct bugs or to add new functionalities. In such systems, the update operation is critical since if an unexpected problem occurs during the updating process, the whole system can be destroyed. This can be the case, for example, during the firmware update of a mobile phone; if connection is lost during the update, then the device cannot be recovered. When performing an update on an embedded application, several problems may appear: power failure during upgrade, bad new/old firmware, communication errors, or flash corruption. Several methods are used to ensure the reliability of updates on embedded systems. The method introduced in [66] is based on always keeping in Flash the firmware that was written before the device was sold or put in function. Whenever a new firmware is written, this ensures that the bootloader can switch to the initial firmware in case the update fails (see Figure 2.3). In case of failure, the watchdog copies the values that indicate a reset because of the new firmware image. The values are checked by the bootloader at system reset and the bootloader chooses the firmware image that should be loaded.
Mobile phone manufacturers and carriers present different software update services that are based on wireless communication such as over-the-air (OTA) updates [71] or distribution centers updates [95]. The idea here is to receive the differential binary (as an update), apply it to the current older binary, and construct the newer binary. Such methods raise concern about the limitations of memory and failure recovery from unexpected situations like a power cut or loss of communication.

**Analyzing user feedback after updating/postactive actions/post-update actions**

One can enhance software updates based on user experiences. Analyzing software behavior after updating helps vendors to improve software evolution. A literature review by Kagdi et al. [67, 32] focused on ”data mining of software repositories” to investigate changes in software components. Another literature review by [16] focused on ”analyzing individual changes” to understand software evolution.

**Preventing failure state/testing before using/proactive actions/pre-update actions**

Software regressions often happen after installing new updates. *Regression testing* is an approach that can be used to avoid such a problem. A well-designed *test plan* can reduce the possibility of the software failure before releasing the software [124]. Therefore, it is recommended to use ”automated testing” and properly written ”test cases” in order
Some research goes further when it comes to updates testing, by allowing end-users to participate in the software testing. A paper by Olivier and his coauthors introduces a framework for "integrating upgrade deployment, user-machine testing, and problem reporting into the overall upgrade development process" [30]. Similar methods are presented in "Efficient Systematic Testing for Dynamically Updatable Software" [55] and also in "Predicting Upgrade Failures Using Dependency Analysis" [28].

2.1.2 Security

While software updating provides a convenient way to fix and enhance software systems, current software update systems have serious security issues [15] that put the software updating process at high risk. Software update mechanisms represent an ideal attack vector for malicious code installations on client machines. In 2006, a paper by Bellisimo, Burgess, and Fu [15] shows variety of possible threats for several software update systems. The threat of the common vulnerabilities in software update systems is more than theoretical. In 2007 a practical research by Francisco Amato introduced the "evilgrade" [5] toolkit that can be downloaded from http://www.infobytesec.com/developments.html. "Evilgrade" provides a framework that can be used to take advantage of poor upgrade implementations to inject fake updates.

With proper understanding of the current software update system vulnerabilities, researchers can improve software update system design to build a secure software update environment that can prevent current threats. A classification of several known security
issues is described in [15, 75, 17, 1, 36, 102, 128, 126]. In this report, we classify the security issues in software updates as follows:

- Authentication and Integrity
- Attacks

**Authentication and Integrity**

Software update authentication is the process of verifying the authenticity of the software update files before downloading or installing the updates on client machines. Without authentication, a software update can be exploited by malicious programs (Trojan horses, viruses, or others) [15, 5, 23]. In the following, we discuss the recent methods and techniques for authenticating software updates.

First, we classify the types of information that software update systems must authenticate during the update process [128, 23]:

- **Content of Updates:** Such content includes update binaries and other related files. An attacker can control clients’ machines by introducing malicious content through the update process (e.g., man-in-the-middle attack). Software update systems must protect clients from installing malicious updates by verifying that all updates are authentic. One of the most used methods to check updates authenticity is by using ”Digital Signatures,” which give evidence that the updates are genuine and were sent by trusted sources that possess proper signing keys.

- **Update metadata:** This include information about the software updates (e.g., the software’s name, version, description, and the necessary dependencies for the updates to run properly). An example of update metadata is the one used by
the Windows Server Update Services (WSUS) [89]. This kind of information is usually contained in XML files or simply in text files (as shown in Figure 2.4).

Update metadata can be used for:

- **Availability of New Updates/Timeliness of Updates:** The first step in the update process is to inform clients about the new updates. This can be done either by announcing the new updates to all registered clients, or by keeping polling update servers for new updates, as in automatic software update systems. If an attacker manages to block out a client from learning about new updates, then the client will continue to use the old version of the Adobe Flash Player 11 Plugin.

![XML code example for Adobe Flash Player 11 Plugin](http://www.w3.org/2001/XMLSchema)
software, which may need installation of security updates. This gives more
time for attackers to exploit known vulnerabilities in the old version of the
software installed on client systems. This kind of attack is called a freeze
attack [128]. In order to prevent freeze attacks, software update systems
must be aware when updates are available. To prevent an attacker from
replaying false responses to the client’s previous queries for updates, clients
must authenticate messages indicating the availability of updates [128].

- Repository State: This metadata contains information about the other
important update contents (the dependencies). For example, a package
manager can learn about the availability of update packages in a certain
repository by downloading the package metadata from that repository. In
addition to that, the package manager learns about updates dependency
information needed to perform ”dependency resolution” [128]. Software
update systems can be tricked by attackers who can use a combination of
metadata and update content which are not stored together on the repository
at the same time. This kind of attack is called a ”metadata inconsistency
attack” [128]. The result of such an attack varies based on the design of
the software update system. The ”metadata inconsistency attack” can trick
a client to allow a malicious version of a dependency to be installed [85,
23]. In order to prevent ”metadata inconsistency attacks,” software update
systems must be able to authenticate information that indicates whether any
subset of downloaded files is from the same repository state. The repository
state is ”a set of files available from a repository at a given time” [128].
In practice, typical software update systems use two primary methods of authentication and integrity: (1) transport layer security (TLS) and (2) digitally signed files [128]. Table 2.1 shows a variety of software update systems and the authentication methods they provide for each type of updates information, as it appears in [128]. Both methods depend on asymmetric cryptography with the idea of using public and secret keys. One of the most important issues in asymmetric cryptography is the key management, which includes: key generation, key distribution and key revocation. A public-key infrastructure (PKI) [93] can be defined as "a system for the creation, storage, and distribution of digital certificates which are used to verify that a particular public key belongs to a certain entity. The PKI creates digital certificates which map public keys to entities, securely stores these certificates in a central repository, and revokes them if needed."

<table>
<thead>
<tr>
<th>Software Update System</th>
<th>Update Content</th>
<th>Update Timeliness</th>
<th>Repository State</th>
</tr>
</thead>
<tbody>
<tr>
<td>yum (Fedora 10)</td>
<td>signed</td>
<td>unprotected</td>
<td>unprotected</td>
</tr>
<tr>
<td>yum (Fedora 11)</td>
<td>SSL + signed</td>
<td>SSL</td>
<td>SSL</td>
</tr>
<tr>
<td>yum (CentOS)</td>
<td>signed</td>
<td>unprotected</td>
<td>unprotected</td>
</tr>
<tr>
<td>yum (Red Hat)</td>
<td>SSL + signed</td>
<td>SSL</td>
<td>SSL + signed</td>
</tr>
<tr>
<td>APT (Ubuntu)</td>
<td>signed</td>
<td>unprotected</td>
<td>signed</td>
</tr>
<tr>
<td>YaST (OpenSUSE)</td>
<td>SSL + signed</td>
<td>SSL</td>
<td>SSL</td>
</tr>
<tr>
<td>YaST (SUSE Ent.)</td>
<td>SSL + signed</td>
<td>SSL</td>
<td>SSL</td>
</tr>
<tr>
<td>slackpkg (Slackware)</td>
<td>signed</td>
<td>unprotected</td>
<td>unprotected</td>
</tr>
<tr>
<td>Update Engine (Google)</td>
<td>SSL</td>
<td>SSL</td>
<td>SSL</td>
</tr>
<tr>
<td>Omaha (Google Update)</td>
<td>SSL + Authenticode</td>
<td>SSL</td>
<td>SSL</td>
</tr>
<tr>
<td>Omaha with CUP</td>
<td>CUP + Authenticode</td>
<td>CUP</td>
<td>CUP</td>
</tr>
<tr>
<td>Adobe AIR applications</td>
<td>SSL + signed</td>
<td>SSL</td>
<td>SSL + signed</td>
</tr>
<tr>
<td>Firefox (Windows)</td>
<td>SSL + Authenticode</td>
<td>SSL</td>
<td>SSL</td>
</tr>
<tr>
<td>Firefox (Mac/Linux)</td>
<td>SSL</td>
<td>SSL</td>
<td>SSL</td>
</tr>
<tr>
<td>Firefox extensions</td>
<td>SSL or signed</td>
<td>SSL</td>
<td>SSL or none</td>
</tr>
</tbody>
</table>

Table 2.1: Software Update Systems and the Authenticated Information They Provide [128]

In the following, we explore research regarding the efficiency of authentication in software updating along with common issues in the key management:
• **Efficient authentication:** A schema for disseminating large updates based on a chain of fragments authenticated efficiently by including the hash of the next fragment in the previously authenticated fragment is proposed in [75]. A similar technique is proposed by Ruopeng Ye in his PhD dissertation “Authenticated Software Update” [126]. Ye introduced an efficient authentication mechanism based on a single digital signature for a bundle of software update packages (instead of signing each package in the bundle). The link between packages within a bundle can be managed by using incremental fingerprint algorithms, which allow for bundling partial updates (only affected packages).

• **Key Management:** As mentioned previously, the key management deals with three aspects: key generation, key distribution and key revocation. Each of these aspects has its own vulnerabilities that should be prevented.

  – **Key generation threats:**

  Generating cryptographically secure keys is an important issue in key management. Cryptosystems use random numbers for the key generation process. "Generating truly random numbers is a difficult problem” [77]. Therefore, various algorithms and techniques are used to solve this problem such as: a random number generator (RNG), pseudo-random number generator (PRNG), or Pretty Good Privacy (PGP). Misusing such algorithms may lead to creating keys that are not sufficiently random. This can help an attacker to predict those keys, and finally, he can decrypt information encrypted with those keys. One good example of misusing the key generation algorithms is in PGP v5.0 [24] which had a problem when generating keys non-interactively that led to weak key generation [77].
– *Key compromising threats:* It is important for software update systems to have enough defense against key compromise. If an attacker manages to obtain private keys that are used for providing software updates, he can use these keys for signing malicious updates. This can put millions of software update clients at risk [128].

Recent research supports the fact that using more than one key can help to improve the security of update systems against attacks [14]. The observation was that when updates are signed with several keys, the work of the attacker is more difficult than for breaking a single key. The solution proposed in [128] generates the various keys starting from a root key, and the whole process is executed under the control of one entity. An attack against this entity or against its root key is still able to compromise the whole system, and its suggested mechanism to minimize risks consists of storing root keys on offline computers [128]. The implementation suggested in [128] is to use separate keys for various roles, such as: the content of updates (target role and delegated target role), the availability of updates (timestamp role).

Another study focusses on reducing the risk of key compromises in process control systems using wireless sensor nodes [102]. That technique is about asynchronously rekeying secure communication for updates.

– *Key revocation threats:* Many update systems have problems revoking a compromised key and certifying a new one. Even if the software vendors find out a key is compromised, there is no secure means of key revocation. For example, in 2008, popular Linux distributions like Red Hat and Fedora, which are used by millions of users, have suffered due to
key compromise [109, 146]. An article in 2009 by Moxie Marlinspike [84] exploits issues regarding Online Certificate Status Protocol (OCSP), which is an important protocol for certificate revocation in PKI structure. The article shows that an OCSP can be defeated by responding to OCSP request with the "tryLater" response-code, which is number 3 in the ASCII character codes. The solution of such a threat has been addressed in [93, 128] which provide techniques for both explicit (CRL/OCSP-based) and implicit (expiration date-based) revocation.

- **PKI vulnerabilities:** Many software update systems use PKI for Transport Layer Security (TLS), Secure Sockets Layer (SSL), and for signing the updates. There are many known threats to PKI [84, 140]. Some of these threats came from the method used for issuing digital certificates like using cryptographically broken algorithms. For example, a practical study in 2008 [140] shows that some trusted Certification Authorities (CA) still use MD5 (cryptographically broken since 2004) and therefore they could manage to introduce a rogue Certification Authority (CA) certificate trusted by most common web browsers.

**Attacks**

Attackers have more time and tools for offense than we have for defense. Here we explore a set of attacks against update systems:

- **Man-in-the-middle (MITM) attack [15]:** when the entire conversation between a client and an update server (mirror) is controlled by the attacker.

- **Arbitrary package [23]:** replace a legitimate package with an attacker package. This leads to controlling the client machine by the attacker.
• **Replay attacks [23]:** send an older package instead of a new one. This can lead to preventing the client from knowing about the new update, which is considered to be "freeze attack."

• **denial of service (DoS) attack:** can be used to prevent or delay of update distributions.

• **Freeze attacks [128]:** block information about new updates. Attackers can use *replay attacks* or *DoS attacks* in order to prevent client from knowing about the new update.

• **Metadata inconsistency attack [128]:** attacker can induce the user to download an extraneous dependencies package.

• **Endless data [23]:** crash the client system by sending him a huge file as an update.

• **Byzantine failures [65]:** when an attacker can be in control of one or more mirrors.

• **Impersonate update servers:** several technique for an attacker to pretend to be a legitimate update server such as DNS cache poisoning [38], BGP prefix hijacking [104] or acting as (or compromising) a legitimate content mirror [23].

Table 2.2 shows common software updating attacks along with the attack cause and the research paper addressing the attack.
<table>
<thead>
<tr>
<th>Purpose of the attack</th>
<th>Attack</th>
<th>Target</th>
<th>Article</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prevent or delay of updates distributions</td>
<td>- MITM attacks</td>
<td>clients</td>
<td>[15, 23, 128, 65]</td>
</tr>
<tr>
<td></td>
<td>- Replay attacks</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Freeze attacks</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Endless data</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Byzantine failures</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- DoS attacks</td>
<td>mirrors</td>
<td>[132]</td>
</tr>
<tr>
<td>Replace legitimate update files with attacker files</td>
<td>- MITM attacks</td>
<td>clients</td>
<td>[15, 65, 23]</td>
</tr>
<tr>
<td></td>
<td>- Metadata inconsistency attack</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Arbitrary package</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Byzantine failures</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Impersonate update servers</td>
<td>- DNS cache poisoning</td>
<td>mirrors</td>
<td>[104, 38, 23]</td>
</tr>
<tr>
<td></td>
<td>- BGP attacks</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Acting as a legitimate mirror</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Compromising a mirror</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: Software Update Common Attacks

2.1.3 Availability

In general, the software update process for client systems starts by stopping the software to be updated, installing the updates, and restarting the software (in some cases the whole system should be restarted). This kind of update process cannot be tolerated
by some systems where the cost of downtime can be prohibitive (e.g., banking, telecommunication, air-traffic controllers and life-support systems). Dynamic software updating (DSU) is a field of research that focusses on updating programs while they are running [107, 96].

Researchers have developed a wide variety of approaches and systems for DSU. Most DSU systems are implemented and tested on real-world programs but none of these systems are considered practical. More research is needed for related issues like: flexibly, security, efficiency and usability of DSU. In this section, we briefly give an overview of the most well known systems and techniques for implementing DSU. These DSU systems can be divided based on the targeted environments:

**Real-time and Embedded systems**

Dynamic update of the software of real-time embedded systems faces numerous challenges: "such systems must meet hard deadlines, cope with limited resources, and adhere to high safety standards" [151]. The studies in this field focus on meeting these constraints (time, limited resources, and safety standards) while updating in runtime [151, 133].

**Programming Languages**

Several systems and tools have been implemented to help programming languages to support DSU. Most of these techniques require developers to specify the locations or points of the updates in their programs in advance. The following is a list of the most used systems in this field:
• OPUS [4] is a DSU tool for applying security patches (not other updates) on C programs. It is considered the first DSU system applied to real-word C programs [139]. OPUS is a viable alternative to the traditional security patching methodology. It uses binary rewriting to apply updates.

• Ginseng [96] is one of the early DSU systems that permits updating active functions on the stack. Programmers are required to set update points in the code which then can be used for dynamic updating. Ginseng provides static analysis tools to help programmers choose the update points.

• PoLUS [25] introduces a new approach that accept updates immediately as they become available, which is different from the previous DSU systems that wait for safety constraints to be satisfied (compared with Ginseng).

• UpStare [80] uses a stack reconstruction mechanism to implement updates. UpStare requires developers to specify a mapping between any possible stack frames.

• Ekiden [56] is implemented for programs written in C. With Ekiden, programmers can export an API that can be used to enable dynamic updating in their programs.

• Kitsune [139] is used for updating programs written in C language. It is “implemented as a runtime framework, a source-to-source translator, and a transformation code generator” [139].

• Rubah [116] is used for updating programs written in Java language. Rubah can dynamically update Java programs that runs on stock JVMs. It supports program
changes in different levels such as changes to the class hierarchy or updates to running methods.

Database Systems

Most of the research efforts in DSU have been focused on updating applications and operating systems. However, as a result for using embedded databases in many applications (e.g., embedded SQLite), DSU systems must support dynamic update of the embedded database schemas.

A recent work by [154] provided an analysis tool SCVD ("Schema extraCtion and eVolution analysis for embedded Databases") which can help researchers in the field of dynamic database update to extract embedded database schemas and to detect schema evolution. The idea of SCVD is based on two modules: (1) The schema extractor module to extract the database schemas embedded in source code along with their schemas into sql files. (2) The schema differencing module to compare the differences between two sql schema files.

Schemas evolve over time to accommodate the changes in the information they represent. Most of the research in schema evolution focuses on enterprise-class databases (not embedded databases). Researchers have proposed a variety of methods and techniques for schema updates and the impact on applications, such as query rewriting and temporal querying [91, 31], schema versioning and snapshot isolation [117], schema mapping [158], or automatic schema matching [120].

Operating Systems

There are several DSU implementations to support dynamic updates of operating systems. One of the most reliable systems is Ksplice [145]. Ksplice is an extension of
the Linux kernel that dynamically applies security updates (patches) while the operating system is running. However, Ksplice only allows updates that do not make significant "semantic changes" to kernel data structures. An update can be generated by Ksplice after determining the changed code in the source code patch within the kernel. Ksplice can analyze code changes in the Executable and Linking Format (ELF) object code layer, rather than at the C source code layer.

Other systems, like DynAMOS [81], use an efficient dynamic code instrumentation technique called "adaptive function cloning." This technique becomes the base for dynamic update of "non-quiescent kernel subsystems" when the timeliness of an update depends on synchronization of multiple kernel paths.

2.1.4 Usability

The term "usability" is used here to describe the ease of use and learnability of the software update systems. In order to understand the importance of the usability in the update system, let’s look at a real case in the Update Manager for Ubuntu. Figure 2.5 shows a message during the update of Ubuntu that is offering "Partial Upgrade." Such messages were responsible for a lot of questions and threads around the Ubuntu community. The ambiguity in messages during the update process may cause fatal errors and major failure. However, some articles [139, 23] refer to the importance of the usability of the update systems and how it is affecting the update process. For example, [23] suggested that package managers should support additional usability requirements like enabling users to verify standalone packages. The authors of the paper proposed a mechanism based on signatures embedded in packages to provide the ease of use when standalone packages must be verified. A technique by [48] introduced
”non-privileged package management” as a solution for the problem of non-privileged users using package management in Linux, which rely on a centralized model where control lies in the hands of the administrator(s). The non-privileged user must rely on the administrator to install the packages he needs.

Figure 2.5: "Partial Upgrade” Offered When Trying to Update Ubuntu

2.1.5 Efficiency

The performance of software updating can be measured by different factors based on the type of the software and the deployment environment. For example, the performance of upgrading Wireless Sensor Networks (WSNs) can be measured by upgrade time, the number of relay nodes, energy consumption and error rates according to several packet sizes. An energy efficient method is presented in [64] by selecting an optimal relay node in WSNs. Steady-state overhead and compilation overhead are important factors to measure performance in DSU [139].

Some research introduced efficient digital signatures technique to increase the performance of software updating. Two methods are presented in [75, 126] for efficient authentication for updates. "Multi-tiered incremental software updating
technology” [141] is an efficient way of downloading and applying updates incrementally instead of upgrading from scratch. Symantec, Inc. uses this technology for incrementally updating virus definitions database on client machines.

2.2 Updating Open-Source Software

More issues are raised in the process of updating free open-source software (FOSS) compared with proprietary software (close-source) [101, 111, 59, 131, 153, 105, 42, 106, 148]. In this section, we give an overview of the difficulties and problems regarding OSS updating.

- **OSS is developed by volunteers:** Most of the FOSS are developed by volunteer programmers with different incentives (reputation, challenging problem,...) [42]. Volunteer programmers spend most of the time on the development of the software [101] (where the challenge is!) but they give less effort in software testing (less challenging problems). This fact affects the reliability of the software updating. However, some researchers have a different opinion; they believe that both proprietary and open source software have roughly the same reliability issues when initially introduced, but open source tends to reduce the ”defect density” in a much faster way compared with proprietary software. Research by Succi et al. ”finds that the changing rate or the functions modified as a percentage of the total functions is higher in open-source projects than in closed-source projects. This supports the hypothesis that defects may be found and fixed more quickly in open-source projects than in closed-source projects and may be an added benefit for using the open-source development model” [111].
• **OSS security vs. Security through obscurity**: When it comes to security in OSS, some studies have found that OSS has fewer vulnerabilities compared with closed-source software [59]. On the other hand, some studies do not support that idea and declare trust issues in OSS [106, 148]. However, other studies do not think this comparison is applicable [131].

• **OSS distribution does rely on public mirrors**: It is common that OSS uses public mirrors (hosted by outside organizations) to distribute the software. Mirrors provide fault tolerance (if a mirror is inactive then the client can use other mirrors) and offload traffic (bandwidth costs) from the distribution's main repository. Software update systems must be able to recognize malicious mirrors and block out updates from them [23, 65].

• **Several distribution branches for the same FOSS**: Regarding the fact that the source-code of an OSS can be recompiled and redistributed under different branch names (e.g., OSS with GPL license), the update process is more complex when compared to proprietary software.

• **The usability of free and open-source software**: FOSS, just like other software, has issues regarding the software usability. However, open-source software has worse usability in general compared with proprietary software (close-source) [101, 52]. This is because FOSS is developed by volunteer programmers who focus more on the software functionality and the features [101] (where the challenge is!), and do not give much thought to the software ease of use (e.g., user interface). This kind of behavior is part of the nature of many programmers. Eric S. Raymond described how programmers view menus and icons “like the frosting on a cake after you’ve
baked it” [101, 52]. OSS usability issues also include the long time needed (in many cases) to install and maintain OSS in a production environment [49]

### 2.3 Recommender Systems

Our contribution in Chapter 4 of this thesis is related to the recommender systems. Recommender systems have become an interesting research area in both academia and industry [3, 108].

**Definition 1 (Recommender System)** "The Recommender Systems are software tools and techniques providing suggestions for items to be of use to a user” [68].

The term "item" here is used to refer to what the system recommends to users (e.g., the item can be: book, music, hotel, specialist, etc.). These suggestions can be used by users for various decision-making processes, such as what camera to buy, what hotel to stay in, or what book to read.

In the mid-1990s, recommender systems became known as an independent area of research after the publication of the first studies on collaborative filtering [121, 123, 58, 136]. From that time, the research in the field of the recommendation systems become more focused on recommendation problems that rely on the ”ratings structure” [3]. This kind of recommendation systems focus on the problem of ”estimating ratings” for the unrated items (not yet seen by a user), for example, the movie recommender proposed by Konstan [72] and the book recommender proposed by Mooney and Roy [92]. In general, recommender systems can be used for two main purposes [63]. First, they can be used to motivate users into selecting a specific item or doing some action such as listening to a specific song or buying a specific book. Second,
recommender systems can be used as filtering tools to deal with information overload. Such systems can be used to select the most interesting subset of items from a larger set. This concept is also used in the fields of information retrieval and information filtering. Therefore, researchers believe that the roots of recommender systems come from a research study in information filtering in 1987 by Malone et al. [82], who proposed different methods of filtering: social filtering ("collaborative filtering"), cognitive filtering ("content-based filtering"), economic filtering, and a combination of these methods ("hybrid method").

The recommendation problem that explicitly relies on the ratings structure can be formulated [3] as follows:

**Classic Formalization of the Recommendation Problem.** Let \( U \) be the set of all users \( U = \{u_1, u_2, ..., u_m\} \) and let \( I \) be the set of all possible items \( I = \{i_1, i_2, ..., i_n\} \) that can be recommended to users, such as books, images, or movies. Each user \( u_k \in U \) can be defined with a set of attributes (user profile) that can include one or more user characteristics based on the applications, such as user_id, DOB, gender, region, or income. Similarly, each item \( i_j \in I \) can be defined with a set of attributes \( \{A_1, ..., A_a\} \) (item record) based on the application. For example, in a book recommendation system, where \( I \) is a collection of books, each book \( i_j \) can be defined by its ID, title, author, publisher, year of release, etc. The space of \( U \) and \( I \) can be very large (e.g., Amazon.com: "millions of users (customers) and millions of items (merchandise")

Let \( f \) be a utility function that measures the usefulness of item \( i \) to user \( u \),

\[
f : U \times I \rightarrow R
\] (2.1)
where R is the totally ordered set used for scoring (e.g., R can be a set of integers within a certain range). “In recommender systems, the utility of an item is usually represented by a rating, which indicates how a particular user liked a particular item” [3]. For example, in Table 2.3, user $u_1$ gave the item $i_1$ the rating of 4 on a one-to-five scale (here, $R = \{1, 2, 3, 4, 5\}$). Formally, we say, $f(u_1, i_1) = 4 \in R$. Based on the application, utility function $f$ can either be specified by users (e.g., $u_1$ gave 4 as a rate for $i_1$) or specified by the application (e.g., profit-based utility function computed by the application). The goal here is for each user $u \in U$, we want to choose an item $i' \in I$ that maximizes the user’s utility:

$$\forall u \in U, \quad i' = \arg \max_{i \in I} f(u, i) \tag{2.2}$$

The main problem of recommender systems is that utility $f$, in most cases, is not defined for each $(u_k, i_j)$ in the $U \times I$ space. This means $f$ needs to be generalized to the whole space $U \times I$. The utility in recommender systems is typically defined on the items previously rated by the users. An example of a user-item rating matrix for a generic recommender system is shown in Table 2.3, where ratings are specified on the scale of 1 to 5. The $\phi$ for some of the ratings in Table 2.3 (e.g., $f(u_2, i_1) = \phi$) means that the users have not rated the corresponding items. Accordingly, the recommender systems should be able to predict the ratings of unrated items for each user and produce appropriate recommendations.

The process of estimating the ratings for the not-yet-rated items is usually based on 1) identifying various heuristics that define the utility function and empirically validating its performance (e.g., applying experiments) and 2) estimating the utility function (estimator) that optimizes a certain performance criterion, such as the statistical
estimator: mean square error MSE. The output of recommender systems after estimating the ratings can be shown to users in different forms:

a) by selecting the highest rating among all the estimated ratings for a user, or

b) by recommending the N best items to a user or a set of users to an item.

The development of recommender systems is a "multi-disciplinary effort" [68] that uses methods and techniques from various fields such as machine learning, approximation theory, artificial intelligence, human computer interaction, information technology, data mining, statistics, adaptive user interfaces, decision support systems, or marketing.

Recommender systems can be classified based on how recommendations are made [8, 63].

### 2.3.1 Content-Based Recommendations (Content-Based Filtering)

In content-based methods, the recommender systems suggest items similar to those that a user liked in the past [74, 113, 92]. The utility of unrated item $i$ for user $u$ ($f(u, i)$) is predicted based on the utilities specified by user $u$ to $I' \subseteq I$, a set of similar items to $i$ that user $u$ rated in the past. For example, in a book recommendation system, in order to recommend books to user $u$, the recommender system learns about preferences of user $u$ by analyzing the common characteristics and features of the books.
that user $u$ has rated highly in the past (based on authors, genres, topics, etc.). Finally, the system suggests the books that have a high degree of similarity to the user $u$' preferences. As we can see in this example, content-based recommendation is based on the availability of 1) an item record (item descriptions), which can be manually created or automatically extracted, and 2) a user profile that contains information about the user’ tastes, preferences, and needs. The user profile can be created either by explicitly asking users about interests and preferences (e.g., using questionnaires), or implicitly learned by analyzing the users’ behavior and feedback through their transactions over time (a user's history of transactions). Content-based filtering is currently used by many recommender systems that focus on recommending items with textual content information, such as documents [113], Web sites (URLs) [9], and netnews [74].

### 2.3.2 Collaborative Recommendations (Collaborative Filtering)

Collaborative recommender systems estimate the utility of items for a particular user based on the items previously rated by other users who shared the same interests in these items in the past (i.e., it is based on the opinions of other users). The estimation of the utility of item $i$ for user $u$, $f(u, i)$, is derived from the utilities of item $i$ for a set of users $U'$ who are "similar" to user $u$, $U' \subseteq U$. For example, in a book recommender system, in order to recommend books to user $u$, the collaborative recommender system maintains a set of users (known as peers or neighbors) $U'$ that contains like-minded users of user $u$. In this example, $U'$ contains users who have rated the same books similarly (have similar tastes in the books). Finally, the system suggests only the books that are most liked by the peers of user $u$. 
In general, algorithms for collaborative recommendations can be classified into two groups [20]:

1. Memory-based (or heuristic-based)

2. Model-based

**Memory-based Collaborating Filtering**

Memory-based (or heuristic-based): Memory-based algorithms [20, 34, 94, 121, 136] are based on heuristics that use the entire collection of previously rated items by the users for estimating the rates for unrated items. The value of unknown rating for user \( u \) and item \( i \) is \( r_{u,i} \), which can be computed as an aggregate of the ratings of the \( N \) most similar users for the same item \( i \):

\[
r_{u,i} = \text{aggr}_{u' \in U'} r_{u',i}
\]

where, \( U' \) refers to the set of \( N \) users who are the most similar to user \( u \) and who have rated item \( i \), i.e., \( N = |U'| \), \( 1 \leq N \leq |U| \).

The aggregation method can be a simple rating average, as shown in 2.4:

\[
r_{u,i} = \frac{1}{N} \sum_{u' \in U'} r_{u',i}
\]

The most common aggregation method uses a *weighted sum* and a *normalizing factor*, as defined in 2.5.

\[
r_{u,i} = k \sum_{u' \in U'} \text{sim}(u, u') \times r_{u',i}
\]
where, $k$ is used here as a normalizing factor and is usually computed as in 2.6

$$k = \frac{1}{\sum_{u' \in U'} |\text{sim}(u, u')|}$$  \hspace{1cm} (2.6)

The similarity $\text{sim}(u, u')$ in 2.6 is used as a weight for the similarity measure between users $u$ and $u'$. The $\text{sim}(u, u')$ is a heuristic artifact that is used to find a set of nearest neighbors or closest peers for each user and it is used to simplify the computing of the rating estimation. In collaborative recommender systems, there are different approaches to computing the similarity between users $\text{sim}(u, u')$. The ”correlation-based” and ”cosine-based” are the most popular approaches for computing the similarity $\text{sim}(u, u')$. For example, the ”Pearson correlation coefficient” is used as a correlation-based approach in [122, 136], to compute the similarity as shown in 2.7:

$$\text{sim}(u, u') = \frac{\sum_{i \in I_{uu'}} (r_{u,i} - \bar{r}_u)(r_{u',i} - \bar{r}_{u'})}{\sqrt{\sum_{i \in I_{uu'}} (r_{u,i} - \bar{r}_u)^2 \sum_{i \in I_{uu'}} (r_{u',i} - \bar{r}_{u'})^2}}$$  \hspace{1cm} (2.7)

where, $I_{uu'}$ in 2.7 is the set of all items corated by both users $u$ and $u'$, i.e., $I_{uu'} = \{i \in I | r_{u,i} \neq \emptyset \& r_{u',i} \neq \emptyset\}$. The symbol $\emptyset$ in $r_{u,i} = \emptyset$ means that item $i$ has not been rated by user $u$. The average rating of user $u$, $\bar{r}$, in 2.7 is defined as:

$$\bar{r}_u = \frac{1}{|I_u|} \sum_{i \in I_u} r_{u,i}, \text{ where } I_u = \{i \in I | r_{u,i} \neq \emptyset\}$$  \hspace{1cm} (2.8)

On the other hand, the cosine-based approach as in [20, 129] uses two vectors, $\vec{u}$ and $\vec{u}'$, in $m$-dimensional space, to represent the two users $u$ and $u'$, where $m = |I_{uu'}|$. Then, the measurement of the similarity between the two vectors, $\vec{u}$ and $\vec{u}'$, can be computed
by the cosine of the angle between them $\cos(\vec{u}, \vec{u}')$ as shown in 2.9:

$$
sim(u, u') = \cos(\vec{u}, \vec{u}') = \frac{\vec{u} \cdot \vec{u}'}{(||\vec{u}||_2 \times ||\vec{u}'||_2)} = \frac{\sum_{i \in I_{uu'}} r_{ui} r_{u'i}}{\sqrt{\sum_{i \in I_{uu'}} r_{ui}^2} \sqrt{\sum_{i \in I_{uu'}} r_{u'i}^2}} \tag{2.9}
$$

where $\vec{u} \cdot \vec{u}'$ is the dot-product between the two vectors $\vec{u}$ and $\vec{u}'$. Beside correlation-based and cosine-based approaches, there are other approaches like the squared difference measure [136], which can be used to measure the similarity between users.

**Model-based Collaborative Filtering**

Model-based: the algorithms in the model-based approach [87, 20, 46, 47, 60, 83, 112, 149] use a learned model to predict unknown ratings, based on the current collection of ratings. For example, in [20] a collaborative filtering system is proposed based on a probabilistic approach to compute unknown ratings between user $u$ and item $i$ as in 2.10:

$$
r_{ui} = E(r_{ui}) = \sum_{j=0}^{n} j \times Pr(r_{ui} = j | r_{ui}, i' \in I_u) \tag{2.10}
$$

where, $j$ in 2.10 represents the rating values as integers between 0 and $n$ and the probability expression is the probability that user $u$ will give a particular rating to item $i$ based on user $u$ previously rated items $I_u$. This probability can be estimated by two alternative probabilistic models [20]: Bayesian networks and cluster models.

The core difference between heuristic-based and model-based approaches in collaborative filtering systems is that the heuristic-based techniques predict ratings based on some ad hoc heuristic rules, while, model-based techniques are based on a model
learned by using machine learning and statistical techniques that analyze the underlying data (e.g., users’ ratings, users’ preferences, or items’ features).

2.3.3 Hybrid Approaches

The basic idea of these systems is to combine the previously discussed approaches (collaborative and content-based) into hybrid recommender systems with more precise recommendations that overcome the limitations of using a single approach as in [9, 12, 26, 114, 130, 100]. According to [3], there are four different techniques to combine content-based and collaborative approaches into a hybrid recommender system:

1. Combining the separate outcome predictions from content-based and collaborative systems [26, 114].

2. Integrating some content-based characteristics into collaborative models [9].

3. Integrating some collaborative characteristics into content-based models [100].

4. Building a single unifying model based on both collaborative and content-based characteristics [12, 130].

2.3.4 Architecture-based Classifications

Based on the architecture of the network, the recommender systems can be categorized in two groups [159]:

- **Centralized Recommendations:** which are client-server based recommender systems that use a centralized server to interact with clients for gathering, processing, predicting, and storing recommendations. Most existing
recommender systems are based on client-server architecture such as the systems used by Amazon.com, Netflix, YouTube, etc.

- **Distributed Recommendations:** which are peer-to-peer based recommender systems that distribute the *recommendations handling* between peers in the network. Each peer is equipped with a distributed recommender agent that receives the underlying data (e.g., user ratings, user profiles, and item attributes.) supplied from the source peers (neighbor peers), and delivers data to the destination peers (neighbor peers) after some certain processing or decision making [159]. One example of such systems is proposed in [53, 157] which use a distributed collaborative filtering algorithm (PipeCF) based on DHT technique to build a scalable distributed recommender system. Another example of a distributed collaborative filtering algorithm is the one used by the PocketLens system [90], which introduced five peer-to-peer architectures to find neighbors. PEOR is another P2P collaborative filtering-based recommendation system, by Kim et al., for multimedia recommendations [70]. Other research by Rosaci et al. introduced a distributed recommender system (MUADDIB) based on community partitions [125]. Zhen et al. proposed a distributed recommender system for P2P knowledge sharing among collaborative team members [159].

### 2.4 Free Riding in P2P Systems

In a peer-to-peer (P2P) network, a *free rider* is a peer that receives the network services while not contributing to others in the network at an acceptable level [44, 69]. Free riding in P2P networks can adversely affect their proper operation. For example, in a file-sharing P2P system, if the majority of peers do not share files, then with time,
"honest peers” who have contributed many files to the network may eventually leave the system. This can lead to resource starvation and performance degradation [69]. In order for P2P system to be viable, it should eliminate or reduce the free riding impact. This can be done by designing incentives to discourage free riding. Incentives in P2P are an important issue regarding the distribution nature of the P2P model and the problem of free riding. A number of incentive-based systems in a P2P environment have been proposed. For example, Karma [150] is a framework that uses a monetary scheme as an incentive to reduce free riding in P2P systems. Friedman et al. [44] proposed an advertisement mechanism to be used as an incentive in P2P systems to donate resources. Other P2P systems use reputation (e.g., KaZaA [78]) as an incentive for cooperation between peers instead of tit-for-tat strategies [33]. Tit-for-tat strategies cannot be applied in some cases where some peers simply cannot contribute to others because they are not capable. For example, some nodes in an unstructured P2P network can be behind some kind of NATs or firewalls and therefore they need communication services from other nodes (supernodes). This is where other incentive mechanisms can be used to convince supernodes to provide the necessary services.
Chapter 3

Secure Auto-Update for Open-Source Software

3.1 Introduction

An essential feature of any software application is the provision of an automatic update system. Without automatic updates, users are exposed to new and unexpected security attacks. Lack of automatic updates also leads to long term coexistence of many versions, increasing the complexity of ensuring inter-operation. On the other side, updates with erroneous patches have been causing trouble on famous applications like Apple TV [19]. Updates to a lower quality release can significantly decrease the quality of user experience, as applications can no longer be used for certain purposes. An erroneous update can destroy the update system and block any opportunity for a future automatic fix. Attacks on the update system can replace an application with a similarly looking malware and divert any further updates from the original source.
Traditionally software testers announce their results directly to developers and/or publish them in specialty magazines. With automatic updates, this may already be too late. This problem is even more acute in open-source software, since development can be transferred from one group of volunteers to another, leading to a new type of attack "Stacking the Deck Attack" (SDA) which can take several forms:

1. **Passing the management from some developers to others:** If an attacker (e.g., a person, an organization, or a country) manages to transfer open-source management from certain trusted volunteers to other controlled developers, then the users of the software can be attacked via their automatic update systems. These attacks can be so surreptitious that it may take a long time for any user to even realize what is happening. By the time users detect the attack, the whole agent system may be compromised and its data may be irremediably lost.

2. **Changing the general vision of the project:** If a team of developers changes their philosophy for a given application and decides to discontinue a feature that is essential for a group of users, this decision amounts to an SDA attack from the perspective of those users. By the time that a user finds out negative implications of an update, an automatic update system may have already upgraded her to the new release.

3. **Compromising the main repository of the project:** If an attacker manages to tamper with source files in a repository used for open-source development, he can achieve a powerful attack against all new users of the system and all users of a classical automatic update mechanisms. Unfortunately, even volunteers observing it lack a mechanism to warn other users.
In order to reduce the level of trust that users are required to have into updates for open-source software that is maintained by volunteers, we propose an integrated model for development, testing, and secure automatic updates distribution of open-source software. The model yields an architecture for automatic updates based on signed recommendations (reviews) from a user-defined constellation (e.g., fraction) out of a set of testers that the user trusts. The selection of the trusted testers can be based on personal experiences or on recommender systems (Chapter 4).

In the proposed model, the main idea exploits the observation that each given release can be built separately by each independent tester while obtaining an identical binary. Therefore, many testers can independently verify and certify that a given binary comes from the sources that the tester was able to study. Moreover, these testers can attach warnings concerning new usability problems or disappearance of features as well as praises for improvements. Developers are able to provide an ontology defining supported features and qualities of the software, such that testers can provide standard evaluations. This in turn enables users to select a full automation of the upgrading process.

Rather than depending on a single tester/distributor, and on the security of a single (root) key, users can specify a set of available testers of their choice. Further, users may specify that automatic upgrades should be executed only if a certain constellation (e.g., majority) of her trusted testers provides positive reviews of the same new release.

While we assume that the public keys of the testers can be obtained by end users via external secure channels, the framework defines mechanisms to revoke these keys when they are compromised.

Usually the server from which updates will be downloaded are not owned by the same organization/tester that created the updates. These third-party mirrors are owned
by content delivery networks (CDNs) or by volunteers [15]. In our framework, such mirrors can compile together certified reports from multiple testers for the same release. While defense is needed to prevent a mirror from performing attacks such as the “Endless Data Attacks” [23], mirrors do not have to be trusted themselves for the quality of the upgrade.

A reference implementation is provided and is used for providing automatic updates to the open-source peer-to-peer DDP2P agents. (See Appendix A for more details.) After we describe the state of the art in the next section, we introduce in detail the employed concepts. Further, we present the architecture of the framework and we analyze its properties. A description of the current reference implementation is given.

### 3.2 Contributions

This chapter presents different but related contributions towards building a secure update framework for FOSS. The contributions of the chapter are as follows:

- **Defining Stacking the Deck Attack (SDA) in the process of updating FOSS.**
  
  We introduce a new attack, Stacking the Deck Attack (SDA), which can occur during the process of updating free and open-source software. SDA can be classified as a social engineering attack, where an external beneficiary (attacker) employs psychological manipulation of OSS developers to perform certain actions for the purpose of taking control of the OSS development and distribution. According to [Braiker 2004], there are several mechanisms that manipulators can use to control their victims with positive reinforcement including money, gifts, and public recognition.
• **A solution for SDA attack.** We propose a solution for mitigating the SDA attack by inserting independent intermediaries (testers) between the developers and the end-users, intermediaries that can eventually detect and easily warn or hamper the automatic damage of the whole system. Once a Stacking the Deck Attack is detected, other volunteers can start or recommend new development branches (which is possible given that the software is free and open-source).

• **Updates with multiple secret keys.** Our proposal satisfies all the guidelines for security defined in the state of the art (listed in the background Section 2.1.2), including the more recent principle of reliance on multiple secret keys [128]. Moreover, we introduce an additional security feature: namely that the owners of the various secret keys can be independent (which can only makes attacks harder).

### 3.3 Threat Model

In this chapter we discuss the cases in which attackers can perform the following feats:

- The attacker can take the leadership of the development of the project (Stacking the Deck Attack).
- The attacker can take control over the source repository.
- The attacker can take control over mirrors and their secret keys.
- The attacker can eavesdrop, intercept, modify, or inject messages into the communication [37].
- The attacker can take control over the computers and secret keys of certain testers that build binaries and certify the quality of releases.
While our reference implementation adapts common solutions for each of these attacker capabilities, addressing the first of them is the main contribution of our efforts. Note that for software that is not free (GPL-like), a financially strong attacker can legally buy the project and close it.

3.4 Concepts

In this section we explore/define some important concepts for our proposed framework. We divide these concepts into two different groups: background concepts and introduced concepts.

3.4.1 Background Concepts

The following concepts are common concepts that have different definitions as they appear in literature (state-of-art). These concepts have specific meanings in the proposed framework. Thus, in this section, we define each concept from our view and show how we use them in our solution. We call these concepts background concepts.

**Open-Source Software.** Open source software (OSS) is a paradigm whereby the source code of the applications is made available to testers and users. The open-source paradigm can be used both for commercial and for free software. By revealing their sources, software developers can gain more trust from their users and can get help in detecting and fixing errors. This concept is important, because it is used to enable independent testers to improve the depth of their analysis.

**Free OSS (GPL-like).** There exist a significant number of open-source projects driven by volunteers, where the result of the development can be freely used, distributed, and
extended. The Linux kernel is an example of free open-source software (FOSS). This concept is essential to enable volunteers to branch development if they disagree with the vision of the current development team (or if they suspect a “Stacking the Deck Attack”).

Under GPL-like licenses (GPL, AGPL, LGPL,...), users are allowed, free of charge, to use, distribute, change, and extend FOSS [153].

Repositories. Most FOSS development relies on developers who voluntarily contribute their time and effort. Such collaboration commonly employs a system that helps in their coordination. For example, a centralized subversion (CVS/SVN) repository can manage the software development activities: add, delete, and update source files. Volunteers use mailing lists for discussing and coordinating [6]. Among the most famous repositories one finds: github.com, code.google.com, and sourceforge.net. Intermediaries (testers) in our solution take their input from the developers using repositories.

Mirror/Distributor. The binary releases and binary updates are available on various servers, typically called mirrors since currently they simply duplicate data from a given site. In our framework the function of the mirrors is extended. A mirror maintainer, referred to as a distributor, can aggregate quality certificates for a binary update from several independent testers and can certify the information about the location (URLs) of the binaries, into an update descriptor. The update descriptor together with the signed binaries constitute an update package.
If a user trusts a certain tester, this tester points the user to the distributors integrating his reviews. Distributors are motivated to integrate reviews from multiple testers in order to make them acceptable to more users.

3.4.2 Introduced Concepts

The proposed framework uses a number of terms/concepts in particular or unique ways as we introduced them in this section.

Quality Definitions. Often programmers develop software trying to achieve a set of predefined requirements specified in a Requirements Document. In the proposed approach, for each new release, programmers provide a standard definition of the claimed qualities of the provided software: Quality Definitions (QDs). The QDs specify the software requirements that are considered to be successfully accomplished in this release [39, 101]. Once defined, future releases cannot redefine a previous definition (but rather can add new ones), thereby helping testers to easily warn users of Stacking the Deck Attacks. Namely, in the absence of this procedure, a Stacking the Deck attacker could remove the definitions for the properties that she disables. Further, testers can add their own additional qualities definitions.

Example 1 The DDP2P software has as claimed qualities:

- support of Windows 7
- it is impossible to falsely attribute an item to another agent
- resilience from censorship
- easy to learn and use (usability)
• each agent controls what data it stores and what data it disseminates

• each agent eventually gets the data of interest for her, when the data is disseminated by a directly connected agent

Users can flag some of these qualities with a threshold to block automatic updates when one of them is not sufficiently supported.

**Binary Builder.** A binary builder is a deterministic function

\[ V : (\vec{\Sigma}, \varepsilon) \rightarrow \beta \]

which associates a unique binary \( \beta \) with a given source \( \vec{\Sigma} \) and set of compilation parameters \( \varepsilon \) (compiler version, options, and target architecture).

It is essential for the binary builder to be deterministic in order to guarantee that a digital signature generated by a tester for the result \( \beta \) of her own compilation of sources \( \vec{\Sigma} \) that she analyzed is applicable to the binaries built and distributed by others. If the binary builder is not deterministic, then one cannot aggregate recommendations from multiple independent testers.

**Independency of Testers.** Two testers are called independent of each other if they can review a software release and give different quality scores independently. The proposed framework allows each tester to study a given release independently while obtaining an identical binary (by applying Binary Builder function as mentioned previously). Therefore, it is possible for any tester to review a release and distribute/post her results
independently. This testing mechanism makes it hard for an attacker to control all the possible testers or to find a common testing behavior.

**Testers/Reviewers.** Our framework brings independent testers into the center of the mechanism for ensuring quality of FOSS. Testers perform the following tasks:

- study each release independently.
- provide quality evaluation information:
  1. Quality of Test (QoT): An estimation of the value of the actual testing efforts out of the amount of testing that needs to be carried out for a QD.
  2. Results of Test (RoT): A score value assigned by testers for each tested QD.
- append their signature to the release, signing the data identifying it (version, file names, hashes) and their quality evaluation (QoT, RoT).

**Example 2** For the case in Example 1, for the first QD, support of Windows 7:

- quality of test: the possible values are (empty or 0: not tested, 0.25: only half of the relevant test-cases were run, 0.5: only binaries were tested, 1: binaries were tested and source was inspected). In Table 3.1, tester A has tested only the binaries, so she specified 0.5 as value for QoT on support of Windows 7.

- result of test: the possible values are (0: not compiling, 0.5: executing with flaws, 1: running well). In Table 3.1, tester A has tested the binaries and found some flaws, so she specified 0.5 as value for RoT on support of Windows 7.
Quality Review. The quality review (or certificate) provided by a tester for a given binary release consists of a digitally signed package describing the name of the release; the compilation parameters and target architecture; the names, sizes, and digest values of each file in the binary release; as well as the definition and quality of his own tests and a score quantifying the result of these tests.

Quality reviews are attached with releases to warn users of potential issues and attacks. A user can refuse updates that are not accompanied by quality reviews from her trusted testers.

3.5 Architecture

In the proposed framework, each binary release (update) of open-source software undergoes four processes (see Figure 3.1):
A) Development process. Developers keep improving the OSS by adding new features or solving current faults. They use a centralized source repository and versioning operations (e.g., export, checkout, checkin) to manipulate files and produce the next release candidate [35]. To help testers and users tune their expectations for the new release, developers provide a set of quality definitions (QDs). Information about the latest release candidate, including version number, releasing date, source code, and QDs is always available to users, testers, distributors (and the general public) in the source repository.

Formally, the output of the development process is the tuple \( \langle \vec{\Sigma}, \nu, \vec{\Phi}, d \rangle \) where \( \vec{\Sigma} \) stands for the release sources, \( \nu \) is the version identifier, \( \vec{\Phi} \) represents the quality definitions, and \( d \) is the release date. The development process can also recommend a set of compilation parameters \( \varepsilon \) for various targeted systems.

B) Testing process. Testers use the source repository to export (download) the source code of the new update. They are expected to perform the necessary testing based on the QDs provided by the OSS developers. They can also test additional properties (based on their judgment). Such tests are made to inform the users (and
implicitly developers) about the qualities of the release. As a result of this process, testers provide both an assessment of the Quality of Tests (QoTs) and a report on the Result of Tests (RoTs).

Each tester has the freedom to only test a subset of the specified QDs. For example, a security specialist tester may want to only test properties related to security (see tester C in Table 3.1). Similarly, a tester specialized in Linux can test properties related to Linux (see tester B in Table 3.1). Each tester compiles and builds her own binaries from the source by using the binary builder function as mentioned early in the concept section. This guarantees that the binaries she signs are the ones corresponding to the source that she inspects. The tester certifies the binary update by providing a digitally signed package with the necessary information such as version number, releasing date, QDs, and her QoTs and RoTs.

Formally, the output of the testing process is a tuple \( \langle \tau, \beta, \vec{\eta}, \nu, \varepsilon, \vec{\Phi}, \vec{\Upsilon}, \vec{\Theta}, \vec{\Psi}, d, S \rangle \) where \( \tau \) is the ID of this tester, \( \beta \) is the binary software, \( \vec{\eta} \) is a set of files information including (file names, size and hash values of files content), \( \varepsilon \) is the release building parameters (target architecture and compiler version and options), \( \vec{\Upsilon} \) is the set of additional quality definitions added by this tester, \( \vec{\Theta} \) is the vector of Qualities of Tests, \( \vec{\Psi} \) is the vector of the Result of Tests and \( S = SIGN(\delta, \langle \vec{\eta}, \nu, \varepsilon, \vec{\Phi}, \vec{\Upsilon}, \vec{\Theta}, \vec{\Psi}, d \rangle) \) is the associated digital signature created with the secret key \( \delta \) of the tester.

A tester can issue a review based on her study of the source code of the OSS. Such a review is applicable to any \( \varepsilon \), in which case it is issued with a special value for \( \varepsilon \), \( \varepsilon = \perp \) (empty). The signature in this case is computed for \( \vec{\eta} = \perp \).

\( S = SIGN(\delta, \langle \perp, \nu, \perp, \vec{\Phi}, \vec{\Upsilon}, \vec{\Theta}, \vec{\Psi}, d \rangle) \)
C) Integration process. A mirror maintainer integrates $\beta$ as obtained from a tester with the quality reviews, each of type $(\tau_i, \beta, \nu, \varepsilon, \vec{\Phi}, \vec{\Phi}_i, \vec{\Theta}_i, \vec{\Psi}_i, d, S_i)$, from $n$ different testers for the release candidate $(\nu, \varepsilon)$ or $(\nu, \perp)$ into a single update/release package, where $i$ is used to enumerate the available testers. If a tester issues reviews both for $(\nu, \varepsilon)$ and for $(\nu, \perp)$, then only the one for $(\nu, \varepsilon)$ is kept. This integration improves both OSS quality evaluation and end-user security. Each tester has signed the new release information and evaluation and this signature is part of the integrated update/release package. Finally, mirror maintainers make the release package available via their distribution channel (e.g., mirror servers, CDs).

Formally we describe the release package with the tuple $\langle \beta, \nu, \varepsilon, \vec{\Phi}, d, \Gamma \rangle$ where $\Gamma$ is a set of tuples $\{\langle \tau_i, \vec{\Phi}_i, \vec{\Theta}_i, \vec{\Psi}_i, \epsilon, t, S_i \rangle\}$, $\tau_i$ is the ID of tester $i$, $\epsilon$ is a Boolean specifying whether the review is issued for $\varepsilon = \perp$, $t$ is the date of the test data, $\vec{\Theta}_i$ is the Quality of Tests vector from tester $i$, and $\vec{\Psi}_i$ is the Result of Tests vector from tester $i$.

D) Update/Install process. A client keeps polling his trusted mirrors for new updates. If a new update $(\nu, \varepsilon)$ is available at a mirror $m$, then its information and associated quality reviews in $\langle \vec{\eta}, \nu, \varepsilon, \vec{\Phi}, d, \Gamma_m \rangle$ are downloaded from all mirrors where it is available. All the available $\Gamma_m$ from all mirrors $m$ are integrated into a single set of quality reviews: $\Gamma = \bigcup_m \Gamma_m$. The quality reviews in $\langle \vec{\eta}, \nu, \varepsilon, \vec{\Phi}, d, \Gamma \rangle$ are then evaluated. If automatic updates are enabled and user-defined criteria concerning required tester support and minimal quality levels are satisfied, then the binary is downloaded, authenticated, and installed. Any user $u$ can specify complex criteria for triggering automatic acceptance of a new update package, such as the
special constellation of testers and QoT/RoT values, of which a \( (t_u, n_u) \) threshold scheme for trusting any \( t_u \) out of \( n_u \) user-selected testers is just a special case. If automatic updates are disabled, users can inspect the quality reviews and make their decision.
Algorithm 1: End-user Algorithm for Accepting Automatic Updates

1 function evaluateUpdates(⟨\vec{\eta}, \nu, \varepsilon, \vec{\Phi}, d, \Gamma⟩, ⟨w, c, \mu⟩) → Boolean
2 if (\nu not newer than currentVersion) then
3    return false;
4 total_wt ← 0;
5 cnt_testers ← 0;
6 crt_QoT ← [0, ... 0];
7 crt_RoT ← [0, ... 0];
8 remove double occurrence of testers in \Gamma (prefer occurrences with newer date t
and more specific, \varepsilon \neq \bot);
9 foreach (⟨\tau, \vec{\Upsilon}, \vec{\Theta}, \vec{\Psi}, \epsilon, t, S⟩ ∈ \Gamma) do
10    \varepsilon' ← \varepsilon; \vec{\eta}' ← \vec{\eta};
11    if (not \epsilon) then \varepsilon' ← \bot; \vec{\eta}' ← \bot;
12    if (\tau \notin \tilde{\Delta}) then
13       \Gamma ← \Gamma \setminus \{⟨\tau, \vec{\Upsilon}, \vec{\Theta}, \vec{\Psi}, S⟩\};
14       continue;
15    if (revoked(PK(\tau))) then continue;
16    r ← verify(PK(\tau), ⟨\vec{\eta}', \nu, \varepsilon', \vec{\Phi}, \vec{\Theta}, \vec{\Psi}, d⟩);
17    if r = true then
18       total_wt ← total_wt + getWeight(\tau, \vec{\Theta});
19       crt_QoT ← combineQoT(crt_QoT, \vec{\Theta}, \tau);
20       crt_RoT ← combineRoT(crt_RoT, \vec{\Psi}, \vec{\Theta}, \tau);
21       cnt_testers ← cnt_testers + 1;
22    else
23       \Gamma ← \Gamma \setminus \{⟨\tau, \vec{\Upsilon}, \vec{\Theta}, \vec{\Psi}, S⟩\};
24    if (getRequiredTesters() \not\subseteq \Gamma) then return false;
25    if (crt_QoT \not\geq getRequiredQoT()) then
26       return false;
27    if (crt_RoT \not\geq getRequiredRoT()) then
28       return false;
29    if (\mu = WEIGHT) then
30       return (total_weight \geq w);  
31    if (\mu = COUNT) then
32       return (cnt_testers \geq c);
3.6 Decision Making for Accepting Automatic Updates

In this section we detail the procedure followed by an agent to decide whether to download and install new updates automatically (see Algorithm 1). The function evaluateUpdate() verifies that the conditions set by user for automatically accepting new updates are satisfied and returns true on success. The two parameters used by it are:

- The quality reviews of an update binary release, aggregated in the tuple: \( \langle \hat{\eta}, \nu, \varepsilon, \hat{\Phi}, d, \Gamma \rangle \)

- The user predefined conditions for each quality definition, aggregated in the tuple: \( \langle w, c, \mu, \Delta \rangle \) where \( w \) is the minimum total weight of trusted testers supporting the update, \( c \) is the minimum number of trusted testers supporting the update, \( \mu \) is the method used to evaluate trusted testers (with possible values: WEIGHT and COUNT), and \( \Delta \) is the list of all testers trusted by the user.

After \( \varepsilon \) is found relevant for the current system, the algorithm compares the current software version (\( currentVersion \)) with the newly received update version (\( \nu \)). If \( currentVersion \) is not older, then the update is rejected. The total weight of the trusted testers supporting this update and their count are computed and stored in the variables \( total\_wt \) and \( cnt\_testers \), respectively (Lines 4, 5, 18 and 21). The combined quality of tests and results are maintained in the vectors \( crt\_QoT \) and \( crt\_RoT \) (see Lines 6, 7, 19 and 20). A sample combination function for \( QoT \) is max and for \( RoT \) is min. \( crtWeight \) returns the weight of a tester, given the user configuration and her own evaluation of her quality of tests.

In order to calculate \( total\_wt, cnt\_testers, crt\_QoT \) and \( crt\_RoT \), we need to iterate over all testers in \( \Gamma \) (Line 9). If a tester’s identifier, \( \tau \) (digest of its public key), is not
found in the list of trusted testers, \( \bar{\Delta} \), then its review is excluded from \( \Gamma \) (Lines 12 and 13). The revocation status of the public key from \( \tau \) is checked using available methods, e.g., CRL, OCSP (Line 15). Reviews from revoked or unknown testers are discarded by the continue operation. Reviews from trusted testers are verified using stored public keys (Line 16). This public key is returned by \( \text{PK}(\tau) \). If the signature of the review is not valid, then that review is excluded from \( \Gamma \) (Line 23). Function \text{getRequiredTesters()} returns a list of the testers without whose supporting reviews the user refuses any automatic update (Line 24).

Function \text{getRequiredQoT()} (used in Line 25) returns the vector containing the minimum amount of testing as required by the user for accepting an automatic update. This condition is evaluated in Line 25 where each entry of \( \text{crt}_\text{QoT} \) must be greater or equal to the corresponding required value. Function \text{getRequiredRoT()} (used in Line 27) returns the vector containing the minimum result for each test as required by the user for accepting an automatic update. If any entry in the \( \text{crt}_\text{RoT} \) is smaller than the corresponding entry in the result of \( \text{getRequiredRoT} \), then the update is abandoned. Based on the value of a given \( \mu \), trusted testers can be evaluated either based on their total weight (Line 29) or based on their total number (Line 31).

### 3.7 Updates Protocol

To download updates, a software repeatedly connects to each mirror, \( \text{url} \), from its set of preferred mirrors (\( M \)). On each connection, the software sends a message \text{update request} taking as parameters the tuple \( \langle \text{ClientID}, \text{url}, d, \text{ver}, \text{crt}, S \rangle \) (see procedure \text{onClock()} in Algorithm 3). In our implementation, procedure \text{onClock()} is set to be called at fixed intervals of time. Here \( \text{ClientID} \) is a unique client
identifier (a public key of the user), $d$ is a timestamp specifying the UTC time of the server as estimated by the client at the moment the request is made (see Algorithm 3 - Lines 3 and 11), $ver$ is an optional parameter that can specify the version that is requested and $crt$ is the current version used by the client. When $ver$ is not specified, the default requested version is the newest release available, and the server does not have to provide it when it is not newer than $crt$. The signature $S$ is computed as $S = \text{SIGN}(SK(\text{ClientID}), \langle \text{url}, d, ver, crt \rangle)$, $SK(\text{ClientID})$ being a notation specifying the secret key of the client identified by $\text{ClientID}$.

Also, the server can probabilistically verify the signature in the update_request (see Algorithm 2 - Line 2). To mitigate DoS attacks, the server only answers requests where its address is the same as the $\text{url}$ parameter in the request (Algorithm 2 - Line 5). If a client sends an invalid signature $S$ or invalid parameter $\text{url}$, blocking her IP has to be made only when IP spoofing attacks can be ruled out (e.g., with TCP connections). The mirror server can log requests to enable blacklisting DoS attackers (see Algorithm 2 - Line 6). When the difference between the time at the mirror and the $d$ parameter is larger than a reasonable threshold $\text{max}\_\text{skew}$, then the server replies with a message time_fault passing as parameter its actual time (see Algorithm 2 - Line 7). An honest client can use this answer to correct its estimation of the time at server (see Algorithm 3 - Line 11).

If all the aforementioned tests succeed, then the mirror returns a message update_response that contains as parameter a digitally signed update descriptor (see Algorithm 2 - Lines 12, 18). This update descriptor contains the update package data described in the section Architecture, except for the binary files $\beta$. The client can verify the signature of the mirror using its stored public keys and can eventually deliberate and decide to download the corresponding files $\beta$, based on Algorithm 1.
Algorithm 2: Mirror Server

1. on update_request (IP, (ClientID, url, d, ver, crt, S)) do
   2. if (! probabilisticVerification(S)) then
      3. block IP;
      4. return FAIL;
   5. if (url ≠ myURL) then block IP and return FAIL;
   6. log(ClientID, url, d, S);
   7. if (|d – time()| > threshold) then
      8. send time_fault(time()) to IP and return FAIL;
   9. if (ver ≠ NULL) then
      1. if (having compatible(ver)) then
         2. send update_response(ver) to IP;
         3. return TRUE;
      3. else
         4. send version_absent to IP;
         5. return FAIL;
   6. if (latestVersion() newer than crt) then
      7. send update_response(latestVersion()) to IP;
      8. return TRUE;
   9. send version_absent to IP and return FAIL;

Algorithm 3: Update Client

1. on clock do
   2. for url ∈ M do
      3. d ← skew(url)+time();
      4. send update_request(myID, url, d, ver, crt, S);
   5. on update_response (update_descriptor) do
      6. store update_descriptor;
      7. conds ← getUserConditions();
      8. if evaluateUpdates(update_descriptor, conds) then
         9. Install new version;
   10. on time_fault (url, server_time) do
      11. skew(url) ← server_time + roundtrip - time;
3.8 Data Formats for Information Exchange

In this section, we explore the structure of the data exchange between clients and mirrors. First, we identify the elements in the "Update Descriptor". Then we describe the structure of the messages (request and response) exchanged between clients and mirrors. In the current implementation, the exchanged messages are encoded using standard representations for web services (XML, WSDL, SOAP).

![Figure 3.2: Update Request](image)

The format of the update descriptor defined in section Architecture is detailed in Figure 3.4. We have seen that a tester generates the data: \( \langle \tau, \beta, \vec{n}, \nu, \varepsilon, \vec{\phi}, \vec{\gamma}, \vec{\Theta}, \vec{\Psi}, d, S \rangle \). This data can be encoded in the update descriptor used in the answers of the mirrors. The update descriptor has four sections: version identification, downloadable items, quality definitions, and data provided by testers as shown in Figure 3.4.

In Figure 3.4, the tester identifier \( \tau \) appears within the tag: \(<\text{digestPK}>\). The information about files \( \vec{n} \) is indicated by \(<\text{downloadables}>\). The quality definitions \( \vec{\phi} \) are specified in element \(<\text{QOTD}>\), each quality being described in a sub-element \(<\text{testDef}>\). The data from a tester is included in the tags \(<\text{testerInfo}>\) in the section \(<\text{testers}>\) of the descriptor. The entries of the quality of test vector \( \vec{\Theta} \) appear in the sub-elements \(<\text{QoT}>\) of the corresponding entry \(<\text{test}>\).
of the result of tests vector $\vec{\Psi}$ appear in the corresponding sub-elements $<\text{RoT}>$, their index into the test vector being specified by the $<\text{ref}>$ element. The additional quality definitions $\vec{\Upsilon}$ of this tester appear in the sub-element $<\text{QOTD}>$ of $<\text{testerInfo}>$. The release date $d$ is in element $<\text{date}>$ and the signature $\mathcal{S}$ in the sub-element $<\text{signature}>$ of the $<\text{testerInfo}>$.

The update descriptor provided by mirrors as the answer to an update request (e.g., the update request can be in the form of a web service request as shown in Figure 3.2) has the same structure but can contain multiple $<\text{testerInfo}>$ elements as sub-elements of the $<\text{testers}>$ element (e.g., mirrors’ update responses can be encapsulated in a SOAP envelope and sent to clients as shown in Figure 3.3).
3.9 Reference Implementation

Here we describe issues solved in our reference implementation of the proposed automatic update system for the open source peer-to-peer DDP2P system, which is implemented mainly in Java. See Appendix A for more details.

**Distribution preparation.** While the Java JDK7 compiler implements a *binary builder,*
this does not hold for the corresponding jar archiver since its output depends on the exact compilation time and date and on the timestamps of the class files. Similar issues can occur with any other build processes that include the build host and time in the program, or with the randomized stack protection cookies when using the stack protector features of GCC.

We implement a binary builder based on the Oracle Java compiler and jar tool by adding a pre-processing on the class files and a post-processing on the resulting jar file. The pre-processing sets all the timestamps of all the archived files to a fix value. The post-processing removes from the resulting jar archive file the information concerning the time of the archiving. For example, since jar is a version of the zip tool, our implementation identifies the first occurrence of the compilation time and date as the 4-byte integer starting at offset 10 (starting from 0). Subsequent occurrences are also removed (see the source code of openjdk as provided by Oracle). The subsequent three occurrences of the time in the jar file are identified by searching the time in the jar file. If only three occurrences are found, then they are identified as the remaining timestamps; otherwise, a second method is used. The second method identifies the timestamp by creating a second jar file and comparing it with the first jar file using the script:

```
cmp -l DD1.jar DD2.jar
```

This outputs a text of the type:
Based on this output (with offsets starting from 1) we identify the remaining three outputs as occurring at addresses: 71, 1441062, 1441121 (starting from 0).

**Client and Server Support.** For supporting the server side (mirror), we make available two packages:

1. A SOAP Toolkit for PHP, which is an extension of the NuSOAP toolkit [7]. It is a set of PHP classes used to generate a WSDL document and handle SOAP requests and produce SOAP responses.

2. A Java package to sign SOAP responses before sending them to clients.

Anybody can query the server using the corresponding standard methods for web services.

We also make available a Java package that implements the communication with the web service provided by the server, as defined in Algorithm 3 with the encoding and decoding of the standard messages given in Figure 3.2 and 3.3. Received updates are handled according to Algorithm 1 for automatic installation of updates.
3.10 Conclusions

We address the problem of the free open-source agent software in applications of strategic importance (like petition drives). We are concerned that such software can be the target of Stacking the Deck Attacks from well funded attackers. This attack consists of orchestrating the transfer of the leadership of the development team to people that they control, enabling a subsequent degradation of the software via automatic updates.

The proposed framework introduces a decentralized authority made up of a cloud of independent testers. Each of these testers can have its own base of users that trust her based on various reasons: reputation, personal contact, or based on independent commercial contracts and services.

Each given user can trust multiple testers with various degrees of trust and can flexibly specify required constellations of Quality of Tests and Results of Tests from these testers in order to automatically accept an update. A threshold trust, of any \( t_u \) out of user \( u \)'s \( n_u \) selected testers, is just a special case of the possibilities enabled by the proposed framework.

The \( t \) out on \( n \) threshold signature security, offered to automatic updates by this framework, is stronger than the security notion offered by known updates techniques for OSS managed by commercial entities. That trust is restricted to a fix set of \( n \) public keys, (moreover, based on a single root key whose attack would be disastrous). Meanwhile, the method proposed here for volunteer based OSS allows each user to select its own set of \( n \) trusted testers with independent keys, from an unbounded/open set of volunteer testers.

While the tester does not have to provide either the code or the infrastructure for mirrors and automatic-updates, they can directly provide/sell services to end-users and
mirrors in terms of *quality reviews* for software. These reviews can be automatically checked in the process of automatic updates. Administrators of *mirrors* can pack together reviews from several testers for a given release. Deterministic *binary builder functions* are introduced for this purpose.

The framework enables the developers of the software to provide an ontology to serve as a common language for testers about claimed achieved requirements of the project. In this way, test results from fully independent testers can be automatically combined in meaningful ways for a safe automatic update scheme.

We provide a reference implementation of the proposed mechanism, integrating it into *DirectDemocracyP2P (DDP2P)*, an open-source software developed by volunteers for supporting petition drives (Appendix A). A sample deterministic binary builder is implemented with this system, guaranteeing that any Java archive (*jar*) built by independent testers with the same version of the java compiler from the same source code release is binary identical (condition required for enabling the composition of reviews from independent testers).
Chapter 4

Distributed Recommendation of Testers

4.1 Introduction

Testers play an essential role in our mechanism for auto-updating agent software based on free open-source software (FOSS). In Chapter 3, we introduced the "Stacking the Deck Attack" SDA (where the attacker can orchestrate taking over the control of the update process). As explained in Section 3.4.2, the independence of testers is essential for the resistance to Stacking the Deck Attacks. A centralized recommendation system has to be avoided since it can be taken over by the attacker, just as the development can. The questions here are, without a centralized recommendation system:

(a) how can users know about the available testers and their mirrors

(b) how can users trust these testers
We provide a distributed recommendation mechanism where testers are automatically rated and advertised to other users. The process of building the involved P2P social network of users and testers is outside the contribution of this chapter, and here we assume that it is provided by the underlying application. Due to this assumption, the proposed technique can be readily integrated with peer-to-peer FOSS while for other types of applications, a social network has to be constructed with mechanisms not addressed here. From here on we assume that the addressed application is a peer-to-peer social network. The application where we tested the concepts is DirectDemocracyP2P.net [138].

4.2 Motivation: Desirable Properties

The independence of the testers is the main principle that can help to maximize the security of the proposed system. In this section, we highlight this principle and introduce heuristic rules that can enhance the independence of the testers.

**Principle [Decentralization]** *The recommendation procedure should not be under the control of a strict subset of users.*

Without this principle, an attacker controlling the recommender system can filter only testers that she controls. Even with a decentralized recommender, the recommendation criteria can be exploited to focus on a few testers (since they are few, they are easier to attack). A heuristic to help distribute the trust away from a small kernel is to take into account proximity.
**Heuristic [Proximity]** Give priority to testers that are close to (or far from) the user, in terms of some social network.

Another heuristic gives priority to testers that are used by fewer neighbors, as a mechanism to improve diversification.

**Heuristic [Diversity]** As a heuristic for independence of testers, one can give priority to testers that are used by fewer neighbors, in terms of some social network.

### 4.3 Recommender System Problem

Unlike in the former recommender problem definitions, in our case we have to define the recommendation to a user to satisfy a set of constraints, $C$, based on the recommendations given previously to other users (e.g., to increase diversity). Therefore, the recommendation weight to the current user has to be computed based on a function that takes as a parameter not only the user characteristics and the ones of the proposed items, but also the "history," i.e., set of recommendations given in the past to other users. This history is modeled as a parameter taking values from a domain $A$ that is a subset of the Cartesian product between the set of users and the set of items, namely the set of possible past recommendations of items to users. Note that this model allows each user in the history to be associated with multiple items recommended to him.

$$f : U \times I \times A \rightarrow R$$
where

\[ A \subseteq U \times I \]

For the case when the recommendation function is also informed of the recommendation value that was used in recommending items to the past users, then the history \( A \) has to be defined on the Cartesian product between the set of users, the set of items, and the set of possible recommendation scores (let us say \( \mathbb{R} \)). In this case:

\[ A \subseteq U \times I \times \mathbb{R} \]

For example, assuming you have three users: \( u_1, u_2, u_3, (U = \{u_1, u_2, u_3\}) \) where each wants to buy a shirt, such that they have different shirts, but where the shirts are matched to their eye color. This is the constraint \( c_1 \), the set of constraints \( C = \{c_1\} \). \( u_1 \) has brown eyes, \( u_2 \) has green eyes, and \( u_3 \) has blue eyes (this information is part of each user’s profile). The available shirts are: red/white (rw), blue/green (bg), and yellow/blue (yb) so \( I = \{rw, bg, yb\} \). The first recommendation is made to \( u_1 \) (empty history, \( A = \{\} \)) who is recommended the red/white (rw) shirt with score 50 based on the score matrix given in Figure 4.1.a (score matrix of eye color and shirt color). The second recommendation is made to \( u_2 \) (where history \( A = \{(u_1, rw, 50)\}) \), who is recommended the blue/green shirt with score 60 as in the matrix shown in Figure 4.1.b. The third recommendation is made to \( u_3 \) (history \( A = \{(u_1, rw, 50), (u_2, bg, 60)\}) \), who is recommended the yellow/blue (yb) shirt with score 40 as in the matrix in Figure 4.1.c. Note that if we use a fixed score matrix (where history \( A \) is discarded), then the best shirt for \( u_3 \) eyes would be also the blue/green (bg) one as in Figure 4.1.a, but due to the use of dynamic history where the scores in the matrix are recalculated based on the changes
on $A$, $u_3$ is recommended the shirt yellow/blue (yb) with score 40 as in the matrix in Figure 4.1.c.

### 4.3.1 Difference from the Classic Recommender System Problem

The target of the most widely used recommender systems is to automatically identify the items that best fit users’ personal tastes [68]. Our goal here goes beyond that to recommend items (testers) that satisfy certain properties/constraints (e.g., to increase diversity) that increase the degree of the decentralization of items (testers) in the whole system. In other words, we focus here on the impacts of the recommendations (distribution of items) on the society of users, not only on individuals. Our formalization captures the fact that our evaluation is dynamic with respect to changes in history. Here, the interest of the user coincides with the interest of society.

### 4.4 The Mechanism of Recommending Testers

There are various ways to build systems that recommend testers to end-users. Since in this work we focus on Stacking the Deck attacks, we investigate a distributed
collaborative recommender CF system such that it favors the independence of the testers. As mentioned above, to address the single point of failure presented by centralized systems to the SDA attack, we propose a peer-to-peer based recommender system. This system distributes recommendation management (recommendation function) between expert peers in the network and automatically assigns testers to unsophisticated users. For sophisticated users, the system simply recommends a set of testers but lets them manually select the testers that they use. Each peer is equipped with a distributed recommender agent that receives a tester’s information along with ratings supplied from the neighbor peers. It then delivers the tester’s information along with ratings to the neighbor peers after a local re-rating and decision making (see Figure 4.2).

For peer-to-peer systems, such as DDP2P or PeerSoN [21], there exists an intrinsic social network as defined by the connections of each peer (e.g., peers and constituents in DDP2P or social-links in PeerSoN). This social network is the base of the proposed recommender system, which uses a heuristic neighborhood-based CF technique for rating prediction. In one such scheme, testers used by a peer are recommended to neighboring peers (linked peers) as shown in Figure 4.2.
By default, the recommendation made to a peer for a tester has the weight given by the maximum value among the weights coming on all its links (neighbors), which is a common practice in collaborative recommender systems (agent-estimating ratings) [3]. On the other hand, users can overwrite this default for themselves by increasing or decreasing the weight manually (user-assigned ratings). For sophisticated users, the recommendation is forwarded only if the user manually accepts using the recommended tester. If a sophisticated user decides not to use a tester that was recommended to her, we assume that she does not trust that tester and therefore she will not pass the recommendation on to others. Other mechanisms can be used to limit the propagation of testers and the memory storage of the peers, as is discussed in Section 6.2. Users can define and act themselves as testers, or manually introduce testers they personally know and trust.

4.4.1 Formalizing Heuristic-based Recommendations

Our approach of recommending testers to users is based on distributed heuristic-based recommendations in a P2P environment. In this section, we formally identify the exchanged messages between peers in the network and then describe the processing algorithms and decision making applied by each peer.

Messages Exchanged

To exchange the information about testers between peers, we are now formalizing the concept of the tester item. We assume that global identifiers (GIDs) are used to uniquely specify peers and testers (as done by common peer platforms).
Definition 2 (Tester Item) A tester item is a tuple \( \langle \tau, A, W, d, P, S \rangle \), where \( \tau \) is the GID (public key) of the tester, \( A \) is the address where the tester can be contacted for retrieving her data, \( W \) is the recommendation weight (trust coefficient) associated with \( \tau \) and made by the sender \( P \) (this weight has been automatically calculated by the sender’s agent or manually assigned by the sender), \( d \) is the timestamp of the given weight, \( P \) is the GID of the sending peer, and \( S \) is the digital signature with which the sender authenticates the provided weight: \( S = \text{SIGN}(SK(P), \langle \tau, A, W, d \rangle) \).

Only the newest tester item is stored for a given pair \( \langle \tau, P \rangle \), as per the timestamp \( d \).

Model I: Applying the Proximity Heuristic

At a user, each tester is associated with a weight (trust coefficient). Testers manually introduced by the user are given a fixed weight (e.g., 100%). This weight can decrease with each level of forwarding (using an amortization coefficient). This amortization is a mechanism to implement a version of the aforementioned heuristic, namely of giving priority to testers who are close to the user in terms of the social network (i.e., proximity heuristic). Proximity, in this context, plays the role of the social influence in social networks where such influence can be defined as a user behavior that leads users to adopt items as a result of a friend’s adoption. That is inspired by the fact many users are attracted to items consumed and liked by their friends (e.g., as a result of trusting friends more than strangers). Social influence is increasingly recognized in research literature as an important factor for improving the quality of recommendations in social networks [79, 134].

At a certain time \( t_{p,m} \) (where \( p \) is the name of the peer and \( m \) is a logical time based on its local clock), peer \( p \) aggregates the received recommendations \( \Omega \) and computes the
new weight for each received tester $\tau_i$ based on “Model I” in Equation 4.1. Then peer $p$ updates its recommendation matrix (see Figure 4.3.b).

$$W_{p,\tau,\Omega} = f \cdot \max_{j \in \Omega} \{W_j|\tau_i = \tau_j\}, \quad (4.1)$$

where $W_{p,\tau,\Omega}$ is the estimation of the weight given/calculated by peer $p$ for tester $\tau$ based on the received recommendations $\Omega$. $f$ is the amortization factor, $\{a|b\}$ denotes the set of elements $a$ for which the condition $b$ holds, and $\max(A)$ is the maximum numerical element of the set $A$. In other words, peer $p$ selects the maximum weight for tester $\tau$ received from neighbors of $p$ (source peers) and then reduces the selected weight with $f$ (the amortization factor). Based on this scheme, peer $p$ gives high priority to testers who are introduced by close neighbors and low priority to those who are introduced by far neighbors (close and far neighbors are measured based on the underlying social network). After calculating the new weight for all received testers, peer $p$ will send only its used testers to its neighbors (destination peers).

For example, assuming the amortization coefficient is 0.9 (i.e., the trust coefficient is reduced by 10% for each new link in the chain of recommendation), the obtained recommendation in an agent system as shown in Figure 4.3. As shown, there are five peers (P1,...,P5) that use two testers: T1 and T2. The user of P5 introduces and uses T1 as a trusted tester and she has started giving T1 a 100% as weight. P3 introduces and uses the tester T2, to whom she also assigns a weight of 100%. Both P3 and P5 pass their selected testers information to neighboring peers (destination peers). In Figure 4.3, P3 announces T2 to her neighbor peers P2, P4, and P5. Also, P5 recommended T1.
to her neighbor peers P1, P2, and P3. Based on these recommendations, each peer updates its recommendation matrix and decides which tester to use. For instance, Figure 4.3.b shows the recommendation matrix of peer P5 as it is changed (updated) by time from time:0 to time:3 (green cells in the matrix show testers used by the peer). At time $t_{P5,0}$, peer P5 had only one trusted tester T1 with weight = 100%, i.e., $W_{P5,T1,\Omega} = 100\%$ (manually assigned by P5), where $\Omega = \{\}$ (P5 has not yet received any recommendations from neighbors). Later on, P5 updated its matrix in time $t_{P5,1}$ based on recommendations received from its neighbors $\Omega = \{ (P3, T2, 100\%) \}$. Therefore, the new matrix has a new column T2 and based on “Model I” in Equation 4.1; T2 is recommended with 90%, i.e., $W_{P5,T2,\Omega} = 90\%$ (automatically predicted by the system). At that time, P3 had decided to trust only one tester T1. In time $t_{P5,2}$, P5 received new recommendations and updated its matrix based on recommendations received from its neighbors $\Omega = \{ (P3, T2, 100\%), (P2, T2, 90\%) \}$ (note that $W_{P5,T2,\Omega}$ still equals 90%). P5 keeps using only one trusted tester T1 for all time (time=0 to
time=3). On the other hand, P4 has decided to use T2 as a trusted tester and forward T2’s information to P1. In addition, P1 has evaluated T1 and T2, then decided to use T1 as a trusted tester (P1 had the choice to use T1[90%] or T2[81%] or both). However, P3 has decided to use T1 as a trusted tester beside T2.

**Model II: Applying Diversity Heuristic**

Applying proximity through an amortization factor $f$ as described in "Model I" in Equation 4.1 can lead to a *social filter bubble effect* [99]. The term "filter bubble" is used in social network to refer to users being surrounded only by friends who have similar interests, which leads them to adopt a similar sets of items (in our case, testers). This effectively isolates users from adopting items out of their current social neighborhood. In order to reduce this effect, we enhance our recommendation model to give priority to testers outside a user’s current social neighborhood. Thus, we use a *diversity heuristic* rule as a mechanism to improve recommendation diversification.

Given a set $\Omega$ of $n$ tester items $\langle \tau_i, A_i, W_i, d_i, P_i, S_i \rangle$ received from distinct neighbors of peer $p$, the current weight of the recommendation made to the user of peer $p$ for tester $\tau_i$ is computed as:

$$W_{p,\tau,\Omega} = f \cdot \left(1 - \frac{\# \{j | \tau_i = \tau_j\}}{K \cdot n}\right) \cdot \max_{j \in \Omega} \{W_j | \tau_i = \tau_j\}, \quad (4.2)$$

where $K$ is a factor modeling the trade-off between proximity and diversity ($K \geq 1$, typically $K = 2$).
For example, if all neighbors recommend only one and the same tester, "Model II" in Equation 4.2 recommends that tester with a weight given by half of the maximum weight received from neighbors (when K=2), amortized with the factor $f$.

In another example, if the user has many neighbors and a given tester is recommended by only one of them, the received recommendation is a large fraction, $\frac{K_n-1}{K_n}$, of the weight received from that neighbor, amortized with $f$. This technique also can solve the new-item problem as appears in many recommender systems [54], where the new introduced tester can be distributed with a high weight.

**Note:** With the described mechanism, once a user influences many of her neighbors to switch to the same testers as she selected, then she will be recommended different testers (for diversification), and the recommendation of the original testers is decreased. This still does not lead to an iterative decrease to 0 of all recommendation weights, as some weights are pinned at certain values by users manually setting values for them.

Figure 4.4 shows the new ratings distribution between peers based on the proposed model ("Model II") in Equation 4.2 where a diversity heuristic is considered. Note that, in a certain time $t_{P5,2}$, P5 received new recommendations and updated its matrix based on recommendations received from its neighbors $\Omega = \{(P3, T2, 100\%), (P2, T2, 45\%)\}$. At that time, P2 was using T2 with weight equal to 45% (with the assumption that P2 received one recommendation only from P3 about T2[100%]). However, later at time $t_{P5,3}$, P5 received new recommendations and updated its matrix based on recommendations received from its neighbors $\Omega = \{(P3, T2, 100\%), (P2, T2, 79\%), (P2, T1, 75\%)\}$. As we can see, P2 have changed the assigned weight for T2 from 45% to 79% because P2 calculates the new weight of T2 based on four received recommendations $\Omega = \{(P1, T1, 45\%), (P3, T2, 100\%), (P3, T1, 45\%), (P5, T1, 100\%)\}$ as shown in
Figure 4.4. Therefore, P2 had a new weight for T2, $W_{P2,T2,\Omega} = 79\%$, T2 weight is increased to satisfy the system property diversity because T2 is recommended by one neighbor (P3), while T1 is recommended by three neighbors (P1, P3, P5).

4.4.2 The Frequency of Tester Replacement

Top-N recommendation for a peer $p$ is done by sorting all received tester items in a list called "Known Testers List" ($L_K(p)$) based on their recommendation scores in decreasing order and recommending the top-N items as a list called "Used Testers List," which is defined as $L_U(p) = \{t_1, \ldots, t_N\}$.

Known Testers List

All the testers known by a peer are stored in a list $L_K(p)$, sorted by their weight. Based on the computation model ("Model II") in Equation 4.2, each peer $p$ evaluates
the weight of the testers in $L_K(p)$ based on the recommendations received from it’s links (neighbors) as in Equation 4.3.

$$L_K(p) \overset{\text{sort}}{\leftarrow} \forall \tau \in \Omega, \; W_{p,\tau,\Omega},$$  \hspace{1cm} (4.3)

$L_K(p)$ items are stored as a pair $⟨\tau, W⟩$. Algorithm 4 shows a function, "buildKnownTestersList()", that receives the recently stored tester items $Ω$, the amortization factor $f$, and the modeling factor $k$. It returns a sorted list $L_K(p)$ that contain the testers’ weights in descending order. Each element of the list $L_K(p)$ is represented by a pair $⟨\tau, W⟩$. The weight of each tester is evaluated based on function "calculateWeight()", Line 6, which is described in Algorithm 5.

**Algorithm 4: Algorithm for building "Known Testers" list ($L_K(p)$)**

1. function buildKnownTestersList(scoreMatrix, receivedTesters, f, k) → [...]
2. $L_K(p) \leftarrow [0, ..., 0];$
3. $n \leftarrow \text{length(receivedTesters)};
4. newWeight \leftarrow 0;
5. for $i \leftarrow 0$ to $n$ do
6.   newWeight \leftarrow calculateWeight(scoreMatrix$[i], f, k, n);
7.   $L_K(p)[i] \leftarrow \text{receivedTesters}[i]||\text{newWeight};$
8. return sort($L_K(p)$);
Algorithm 5: Algorithm for calculating the new weight for a given scores of a tester and given parameters (f, k, n) which applies Model II in Equation 4.2

1 function calculateWeight(T_i, f, k, n) → Number
2   countScores ← length(T_i) - emptyElement(T_i);
3   newWeight ← f * (1 - countScores/(k * n)) * maxScore(T_i);
4   return newWeight;

Used Testers List

Each peer maintains a list of used testers, \(L_U(p)\). Advanced users populate this list manually while remaining users get their \(L_U(p)\) list populated automatically by the distributed recommender system. While a regular user does not manually select testers from the recommended list \(L_U(p)\) and does not manually introduce any, the system automatically uses the top N recommended fraction of the recommended testers (in our experiments, we set this fraction to 10 testers, i.e. N=10). For example, in Figure 4.4.b and in time \(t_{P_{5,3}}\) the \(L_U(p) = [T1 : 100\%]\) while \(L_K(p) = [T1 : 100\%, T2 : 67\%]\). Once a user manually selects testers based on the available recommendations, then her list of testers is no longer automatically classified.

The more different tester configurations are adopted by a peer, the higher is the risk that a configuration controllable by an attacker is eventually selected. In order to reduce the number of experienced tester configurations, a peer only switches a single tester at a time and only with a certain probability, \(Pr\). This mechanism increases the stability of the system by reducing the switching frequency of testers in \(L_U(p)\). Formally, each peer \(p\) updates its trusted testers list \(L_U(p)\) based on Equation 4.4. Note that there are two cases of populating the \(L_U(p)\) list: first, when the current used tester list \(CrtL_U(p)\)
is empty (the size of the list is 0) or the \( \text{CrtL}_U(p) \) is not fully populated (the size of \( \text{CrtL}_U(p) \) is less than the threshold \( N \)). In this case the \( L_U(p) \) list is simply populated with the top-\( N \) elements (testers) in \( L_K(p) \). Second, when \( \text{CrtL}_U(p) \) already has \( N \) elements (testers). In this case the \( L_U(p) \) list has the same elements as \( \text{CrtL}_U(p) \) list except that only one tester \( \text{CrtL}_U(p)[N] \) (with lowest weight) is the candidate for being replaced by the highest weight tester \( L_K(p)[i] \) as long as \( L_K(p)[i] \) does not exist in \( \text{CrtL}_U(p) \). Switching testers in list \( \text{CrtL}_U(p) \) is done with a certain probability, \( Pr \) (e.g., if \( Pr = 0.5 \), then the chance of replacing \( \text{CrtL}_U(p)[N] \) with a new tester from \( L_K(p) \) is 50% ).

\[
L_U(p) = \begin{cases} 
\text{topN}(L_K(p)), & \text{if } |\text{CrtL}_U(p)| < N \\
\text{CrtL}_U(p)[N] \leftarrow \frac{Pr}{L_K(p)[i]} \text{ where } L_K(p)[i] \notin \text{CrtL}_U(p), & \text{if } |\text{CrtL}_U(p)| = N 
\end{cases}
\] (4.4)

where \( |\text{CrtL}_U(p)| \) is the size of the \( \text{CrtL}_U(p) \) list, which can be either 0 (for empty list) or less than \( N \) (for partially populated list) or equal \( N \) (for fully populated list). For the first case where \( |\text{CrtL}_U(p)| < N \), \( L_U(p) \) has the top \( N \) elements of \( L_K(p) \), which is defined as \( \text{topN}(L_K(p)) \). On the other hand, in the case where \( |\text{CrtL}_U(p)| = N \), only the last element in \( \text{CrtL}_U(p) \) may be replaced based on probability \( Pr \). Algorithm 6 shows: function “buildUsedTestersList( )” that returns the \( L_U(p) \) list populated with top-\( N \) testers based on Equation 4.4. The function receives the currently used list \( \text{CrtL}_U(p) \), the \( L_K(p) \) list, the list size (\( N \)), and the switching probability (\( Pr \)). Lines 2 and 3 represent Case 1 in Equation 4.4 where the length of the \( \text{CrtL}_U(p) \) list is less than \( N \). Case 2 is represented by the for loop in Line 4 where each element in \( L_K(p)[i] \) is
checked in Line 5. If $L_K(p)[i]$ does not exist in the $CrtL_U(p)$ list, then it can be used as a replacement for the last element in $CrtL_U(p)$ based on probability $Pr$ as shown in Line 6.

Algorithm 6: Building 'used testers' list ($L_U(p)$)

1 function buildUsedTestersList($CrtL_U(p), L_K(p), N, Pr)$ → [...]
2 if (length($CrtL_U(p)$) < $N$) then
3     return topN($L_K(p), N$);
4 for $i ← 0$ to length($L_K(p)$) do
5     if ($L_K(p)[i] \notin CrtL_U(p)$) then
6         switchBasedOnPr($CrtL_U(p)[N-1], L_K(p)[i], Pr$);
7     return sort($CrtL_U(p)$);
8 return $CrtL_U(p)$;

Figure 4.5: Distributed Recommender Model in Each Peer with Example

91
4.4.3 Distributed Recommender Model in Each Peer

The proposed recommender system model has three main modules:

1. Recommendations receiver (listener): It is triggered by messages coming from neighbor peers that contain tester items. When it receives new tester items from a neighbor peer, the module validates the sender’s signature $S$ and filters out duplicated tester items before storing them. Only the newest tester item is stored for a given pair $\langle \tau, P \rangle$, as per the timestamp $d$. Each received tester item from each peer is stored. The output of this module is the received testers information referred to here as $\Omega$. For simplicity, we think of $\Omega$ as a set of tester items and each element of $\Omega$ is represented by $\langle P, \tau, W \rangle$ as in the example in Figure 4.5. Note, $\Omega$ elements have more information such as $A$ (tester address), $d$ (the timestamp of the given weight), and $S$ (digital signature of the sender). The module updates the testers profile by adding new entries for the new received testers or by updating the information of the existing testers. The neighbors profile is used for the sender’s signature validation process.

2. Lists builder: With a certain frequency (e.g., daily) a peer recomputes the weights of each of the received tester items $\Omega$ and builds lists: $L_K(p)$ and $L_U(p)$ as described in Figure 4.5. This module receives $\Omega$ as an input from the "recommendations file" and then uses the procedure ($ListsBuilder()$) illustrated in Algorithm 8 to build the required lists. The procedure starts by extracting three components from the received recommendations $\Omega$:

   1) List of all distinct testers’ IDs in $\Omega$ ($receivedTesters$ list) as shown in Line 2.
   2) List of all distinct peers’ IDs in $\Omega$ ($sourcePeers$ list) as shown in Line 3.
3) Matrix of the received ratings/scores that is represented by a two-dimensional array scoreMatrix as shown in Line 4.

An example in Figure 4.5 shows the receivedTesters list with five testers \((T1, T2, \ldots, T5)\) and the sourcePeers list with seven neighbor peers \((P1, P2, \ldots, P7)\) after extracting them from the received recommendations \(\Omega\). Also the example shows the rating values of all received testers from source peers in a two-dimensional array scoreMatrix[7][5]. Note that scores are represented by a percentage weight value and the \(\phi\) notation is used to show an unrated tester by the corresponding peer (e.g., \(P1\) did not send a recommendation for \(T1\) and therefore the score is \(\phi\)).

Lines 5 and 6 in procedure ListsBuilder() are for initializing the known tester list \((L_{K}(p))\) and used tester list \((L_{U}(p))\). System parameters \(f, k, N\) and \(Pr\) are retrieved from the system parameter file as shown in Line 7 to 10. For our example in Figure 4.5, the parameters are set as follows: \(f = 0.9, k = 2.0, N = 3\), and \(Pr = 0.5\).

In Line 11, the Known Testers List \((L_{K}(p))\) is populated by calling function buildKnownTestersList() as illustrated in Algorithm 4. The function calculates the new weight for each received tester based on our proposed model, "Model II", in Equation 4.2, which is described in Section 4.4.2.

Based on the example given in Figure 4.5, the new weight of tester \(T1\) is computed in peer \(p\) for the current received recommendation \(\Omega\) as follows:

\[
W_{p,T1,\Omega} = 0.9(1 - \frac{4}{2 \times 5})89\% = 0.9 \times 0.6 \times 89\% = 48\%
\]
After calculating the new weights of all testers in the receivedTesters list, the $L_K(p)$ is sorted in descending order based on the new weights. As shown in the in example Figure 4.5, T1 is the second element in the $L_K(p)$ list and is represented as, $T1 : 48\%$

In Line 12, the Current Used Testers list ($CrtL_U(p)$) is retrieved and then used as a parameter in the function buildUsedTestersList() in Line 13 in order to populate $L_U(p)$ list (described in Section 4.4.2). In our example, the UsedTesterList includes the top N testers of the Known Testers List ($L_K(p)$), where $N = 3$. In the automatic recommendation mode, the system adopts testers in the list $L_U(p)$ list for software updates. Each time a new tester is selected, the tester is automatically contacted at the provided address for a list of mirrors using her reviews.

Finally, the sendRecommendations() procedure is invoked in Line 14 to send the new recommendations to current peer neighbors, which is described in the next module, "Recommendations sender".

3. Recommendations sender: After building the Known Testers List ($L_K(p)$) and Used Testers List ($L_U(p)$) in the Lists Builder module, the sendRecommendations() procedure is triggered to sends recommendation messages to the peer’s links (neighbors). Each recommendation message has all testers in the peer’s $L_U(p)$ list as well as a number $k^d$ of other top testers in its $L_K(p)$ list. The sendRecommendations() procedure is detailed in Algorithm 7, which receives the Known Testers List ($L_K(p)$) and Used Testers List ($L_U(p)$) as input parameters. The procedure starts by retrieving destination peers information from the "neighbors profile" as shown in Line 2. Then in Line 3, the recommendation message is built by calling the
"buildRecommendationMessage" function, which combines recommendations of testers in $L_U(p)$ list and a number $k^t$ of other top testers in its $L_K(p)$ list. Finally the procedure sends the recommendation message to each peer in destination peers list as shown in Lines 4 and 5. Figure 4.5 shows the data flow around "Recommendations sender" module. The input data comes from three sources: 1) neighbor peers profile (e.g, PeerID and network address) 2) Testers Profile (e.g, testerID, name and, url address) 3) Known Testers List ($L_K(p)$) and Used Testers List ($L_U(p)$) to build the recommendation message. The output of the Recommendations Sender module is a recommendation message that is sent to neighbor peers.

Algorithm 7: Sending recommendations procedure

1. procedure sendRecommendations($L_K(p), L_U(p)$) do
2.     destinationPeers ← retrieveDestinationPeers(NeighborsFile);
3.     recommendationMessage ← buildRecommendationMessage($L_K(p), L_U(p)$);
4.     foreach peer ∈ destinationPeers do
5.         sendMsgTo(recommendationMessage, peer);

4. System Management: This module allows users to manage the proposed system and change the default settings through the use of options provided by well designed GUI as shown in Figure 4.6. There are two types of users in our system:

   (a) Regular users (unsophisticated users): those who usually keep the default settings in the system and allow the auto-update option which involves: automatic selecting of trusted testers, automatic updating of testers’ information, automatic updating of mirrors information, etc. This
type of user represents the majority of users in our system. Our experiments are based on this assumption.

(b) **Sophisticated users**: those who have enough knowledge of the system to manually change the default settings provided with every install. They make their own decisions regarding new updates, introducing and weighting/rating testers, selecting trusted testers, managing the size of the trusted testers list, managing used mirrors, etc. *Sophisticated users* play two main roles in the system:

1) **Introducing testers**: a sophisticated user $u$ can introduce a new tester $t$ to the system by adding $t$’s information (e.g. ID, name, email, ...) to $u$’s agent database. Then, tester $t$’s information (tester item) is disseminated over the network only if the introducer user $u$ has already adopted/used tester $t$ and manually assigned a weight/rate to $t$.

2) **Rating testers**: sophisticated users can assign new weights for testers based on their experiences. The new weights of the testers can then be distributed to neighbor peers to be considered for the next evaluation of testers.

The default purpose of the proposed system is to target regular users (the majority of users). Therefore, the default settings of the system use automatic updating with parameters (e.g., $UT$ size and updating time) that support the distribution of testers based on the given system properties (e.g., proximity, diversity, and stability). Thus, the resulting system can reduce the chance of controlling testers by attackers and at the same time it is easy for regular users to use. However, for
sophisticated users, the desire for manual rating of testers and changing system parameters is accommodated through the use of options as shown in Figure 4.6.

Algorithm 8: Lists building procedure

1. procedure ListsBuilder(Ω) do
2. \text{receivedTesters} \leftarrow \text{extractTesters}(Ω);
3. \text{sourcePeers} \leftarrow \text{extractPeers}(Ω);
4. \text{scoreMatrix} \leftarrow \text{extractScoreMatrix}(Ω);
5. \text{L}_K(p) \leftarrow [0, \ldots, 0];
6. \text{L}_U(p) \leftarrow [0, \ldots, 0];
7. \text{f} \leftarrow \text{systemParameters.get}_f();
8. \text{k} \leftarrow \text{systemParameters.get}_k();
9. \text{N} \leftarrow \text{systemParameters.get}_N();
10. \text{Pr} \leftarrow \text{systemParameters.get}_\text{Pr}();
11. \text{L}_K(p) \leftarrow \text{buildKnownTestersList(scoreMatrix, receivedTesters, f, k)};
12. \text{CrtL}_U(p) \leftarrow \text{retrieveCurrentUsedTestersList}();
13. \text{L}_U(p) \leftarrow \text{buildUsedTestersList(CrtL}_U(p), \text{L}_K(p), \text{N, Pr});
14. \text{sendRecommendations(L}_K(p), \text{L}_U(p));
4.4.4 Tester’s Life-Cycle

Each tester in the system has a life-cycle that starts by having him introduced to the system by a user and ends when there are no more recommendations of that tester distributed across the system. We can summarize the tester’s life-cycle into three stages:

1) Introducing tester stage: each tester is introduced to the system by one or more users. The tester information is distributed (disseminated) between users only after satisfying two conditions:

   a. The tester is created, weighted and stored in the introducer peer agent. This includes information such as the tester public key (GID), creation date, name, email, reference URL, and manual rating value (tester weight).
b. The introducer peer (sophisticated user) should manually adopt/use the introduced tester in order to pass its information to other users.

2) Tester item distribution stage: tester item is passed from one peer to another in the system and the weight of the tester is changed each time in the receiver peer, based on ”Model II” in Equation 4.2. This weight can increase or decrease to satisfy the system constraints/properties (e.g., diversity, proximity ..).

3) Termination stage: A tester can eventually be discarded by the system if the tester’s expiration date is passed. Each tester is tagged with an expiration date that can be updated by retrieving the tester status from the tester’s reference URL. Also, users discard testers with weight = ⊥, which indicates a bad reputation of these testers.

4.5 Evaluation And Experiments

In this section, we discuss the possible evaluation strategies of the proposed recommendation system and then use a set of experiments to evaluate the performance of the technique.

4.5.1 Evaluation Strategies

Before we answer an important question -How do we evaluate the proposed recommender system?- we first should realize that evaluating recommender systems depends on the goals of the designed system, the system users being recommended to, and the nature of the items being recommended [135]. Therefore, we start this section by discussing the following system goals, users, and items.
1- **Goals**: As we mentioned before, the goal of our system is to automatically recommend testers to users in a way that enhances testers’ independence. Decentralization of recommendations is the main principle of the system and can be achieved through applying multiple heuristics such as proximity, diversity and stability.

2- **Users**: In our system, the majority of the users are *regular users* who do not manually rate testers; instead they adopt testers that are automatically recommended by the system. The reason for such behavior is that most of the expected users simply prefer using the default setting of the system and they do not have enough experience to rate or introduce testers. On the other hand, only a few users, *sophisticated users*, can introduce and manually rate testers.

3- **Items**: ”Testers” are the recommended items to users in the proposed system. Unlike other recommender systems, like movies recommender systems where any user can rate a movie after watching it and then decide whether he likes it or not (using a scale, e.g., 1 to 5), a tester in our system can be rated only by a sophisticated user who can study that tester by experience and then assign a fair rate based on his opinion.

Most recommendation evaluation techniques are focused on measuring recommendation accuracy [135]. There are various metrics for measuring the accuracy of the system’s recommendations, like the popular ”precision and recall” metrics [57], which show how precise the system is in estimating an item ratings. Most of these measuring methods depend on comparing the system’s automatic ratings with users’ manual ratings of a similar set of items. However, based on the type of users and the nature of the testers in our system, measuring the
accuracy of recommendations can be applied only to the sophisticated users who can receive system’s automatic recommendations and at the same time they, can manually rate testers, select used testers and introduce new testers. Sophisticated users are only represent around 10% of all users. In order to measure accuracy in our system, we can use the popular ”precision” metric [57] for measuring the top N recommendations accuracy for sophisticated users in the system. ”Precision is defined as the ratio of number of relevant items selected to number of items selected” [57]. For our system, the precision metric can be formally written as in Equation 4.5.

\[
\text{precision} = \frac{\sum_{s \in S} |\text{correct}(L_U(s))|}{\sum_{s \in S} |L_U(s)|},
\]

(4.5)

where \(S\) is a set of sophisticated users in the system. \(\text{correct}(L_N(s))\) is a list of all testers who have been ”adopted” by a sophisticated user \(s\) (i.e., testers in the current used list of user \(s, L_U(s)\)). At the same time, these testers have been automatically recommended by the system as the top N testers in the Known Testers List, \(\text{top}_N(L_K(s))\). Formally this can be written as \(\text{correct}(L_N(s)) = \{t \in L_U(s)|t \in \text{top}_N(L_K(s))\}\). In this case, the precision metric shows the probability that a selected tester is relevant.

On the other hand, the recall metric can be defined as ”the ratio of relevant items selected to total number of relevant items available” [57]. Formally, this can be written as in the Equation 4.6. In this case, the recall metric shows the probability that a relevant tester will be selected.

\[
\text{recall} = \frac{\sum_{s \in S} |\text{correct}(L_U(s))|}{\sum_{s \in S} |\text{top}_N(L_K(s))|}.
\]

(4.6)
Note that these accuracy-measuring techniques cannot be applied to regular users who represent the majority of users in the system. This is because regular users always adopt the recommended top N testers, i.e., \( L_U(s) = topN(L_K(s)) \). Therefore, the value of precision and recall is always 1 (note, we ignore the stability constraint for simplicity). However, relying on the accuracy of recommendations between sophisticated users alone is not enough to find the most relevant testers for users. Based on our goals for the proposed recommender system, recommendations do not need to be only accurate, but it is important for them to be useful for the application [57, 86, 135]. Therefore, we consider measures of recommender system usefulness that move beyond accuracy to recommend testers that meet the desired properties (e.g., diversity and stability), which eventually increases the degree of the decentralization of testers in the whole system. In other words, we focus here on the impacts of the system recommendations on the society of users, not only on individuals.

The metrics used to evaluate the performance of each version (e.g., set of parameters) are:

- **Distribution.** The distribution (usage) of testers among peers should be correlated to their perceived quality (as per an aggregated value of the weights provided by the peers manually introducing them). However, the differences should not be large, to avoid giving overwhelming power to attackers that succeed in manipulating their weight.

- **Diversity of Recommendations.** Among many different aspects that cannot be measured by accuracy metrics alone, we focus on the diversity of recommendations. In general, there are two ways to measure the diversity of recommendations: individual and aggregate [2]:

102
1- Individual diversity: In this case, we try to avoid recommending *too similar* items to the same user [2]. It is popular to measure "individual diversity" by calculating the average of the "dissimilarity between all pairs of recommended items" [8], [32], [46], [54], [57].

2- Aggregate diversity: Unlike individual diversity, we try here to measure the degree of diversity of recommendations across all users. In this case, we examine the impact of the proposed recommender systems on testers diversity by considering aggregate diversity [2].

For our experiments, we focus on *aggregate diversity* while we discuss some issues regarding "individual diversity" in the future work Section 6.2. There exist different metrics for measuring *aggregate diversity* such as *coverage* [57]. Coverage can be defined as the "percentage of items that the recommender system is able to make recommendations for" [2]. In our case, we want to measure the performance of the recommender system based on the Used Testers Lists ($L_U(p)$) that have been adopted by system users. Therefore, we use the total number of distinct testers adopted across all users to measure the aggregate diversity. Aggregate diversity can be formally written as in Equation 4.7:

$$aggregateDiversity = \left| \bigcup_{u \in U} L_U(u) \right|, \quad (4.7)$$

where $U$ is a set of all users in the system. $L_U(u)$ is a list of used testers adopted by user $u$. For example, a very low aggregate diversity indicates that all users are being recommended the same top N testers, whereas a very high aggregate diversity points to the fact that every user receives her own unique top N testers.
• **Local Stability.** The set of testers used by a given peer should be stable (not changing frequently). If this set changes frequently, then this increases the likelihood that an attacker eventually gets the opportunity (favorable configuration of testers) to manipulate a given peer into updating to a doctored version. The local stability evaluates the frequency of switching testers for individual peers.

• **Global Stability.** We want to avoid having high local stability for some peers while other peers switch their testers frequently. We measure the global stability as the median of the local stability metrics for all peers.

• **Casualties.** We want to reduce the number of peers taken over by the attacker. This number is computed by counting how many peers eventually use a configuration of testers that can be controlled as whole by the attacker (e.g., the majority of the testers are controlled by the attacker).

Once a peer trusts a set of testers controlled by the attacker, it will download a doctored version of the software (being a casualty) and is itself taken over by the attacker. In theory, the attacker can control the recommendations made by this peer. However, based on the recommendation strategy we use here, it is not trivial whether an attacker is better off recommending controlled testers or un-controlled testers (recommending controlled testers can induce the neighbors peers to avoid them, since now those peers are too frequently used; on the other hand, recommending un-controlled testers wastes opportunities to disseminate controlled ones). The recommender system can be designed to be independent of the rest of the application being updated, thereby making it immune to the *stacking the deck* attack.
4.5.2 Experiments

In order to evaluate the performance of our distributed recommender system for testers, we simulate instances with the following characteristics:

- 10000 peers ($P_0..P_{9999}$) divided into 100 neighborhoods (of 100 peers each). For example, $P_0..P_{99}$ are in neighborhood $N_1$, $P_{100}..P_{199}$ in neighborhood $N_2$, etc.
- Each peer is linked to 50 other peers; 48 of these links are within the neighborhood of the peer (only two of the links are out of the neighborhood).
- 100 testers ($T_0..T_{99}$), each introduced by a peer as follows: $P_{100}$ manually selects $T_i$ for usage with weight 100%.
- Each peer not manually selecting its testers automatically uses at most 10 testers (i.e. $N = 10$), as per our mechanism.
- We evaluate the behavior for multiple values of the parameters $f$, $k$, and $p$.

Our simulation works in rounds (intuitively the decisions taken in one round corresponds to the decision taken at a fixed interval by each peer, e.g., each day). In each round of the simulation, all peers synchronously evaluate recommendations from their links and decide new usage. We ran 100 such rounds for each experiment and then analyzed the results.

4.5.3 Evaluation of Distribution

In the first reported experiment, we show the distribution (usage) of testers among peers after running the simulation with 100 rounds. In this experiment, each peer switches to the top 10 testers recommended to it in the round. The other parameters are $k = 2$ and
As one can see in Figure 4.7 the standard deviation for tester usage is 331. For example (in the worst scenario), assuming an attacker peer introduced the malicious tester $T_{21}$, one ends up with 2803 peers using $T_{21}$ (maximum usage in this experiment).

We also ran an experiment with $p = 5\%$, $k = 2$, and $f = 0.9$. One can see in Figure 4.7 that the standard deviation for tester usage is reduced to 173. Now (in the worst scenario) when an attacker peer introduces a tester $T_{59}$, then $T_{59}$ is used by 1393 peers (maximum usage in this experiment), which reduces its negative effects by more than 50%. Also, the usage of each tester tends to be distributed over several neighborhoods. For example $T_{86}$ is used by 846 peers where only 50 peers are from the same neighborhood $N_{86}$.

### 4.5.4 Global Stability

In this experiment, we first compute the local deviation between testers’ weights in the previous list of used testers ($L_U(p_j)_{time(k-1)}$) and the new one ($L_U(p_j)_{time(k)}$) for peer $p_j$ as the following:

$$\left| L_U(p_j)_{time(k)} - L_U(p_j)_{time(k-1)} \right|.$$
Figure 4.8: Global Stability: where $k \in \{1.1, 2\}$ and the probability of replacement is $P = 5\%$ [comparing parameter $f = 0.9$ and $f = 0.7$]

Then we compute the global deviation by summing up all local deviations for each peer $p_j \in P$, where $P$ is a set of all peers in the system, as in Equation 4.8. For example, let’s assume that the previous list of used testers for peer $p_j$ is as follows:

$\{L_U(p_j)_{\text{time}(k-1)} \mid (T1 : 40\%), (T2 : 70\%), (T3 : 20\%)\}$

and the new list is as follows:

$\{L_U(p_j)_{\text{time}(k)} \mid (T1 : 50\%), (T2 : 60\%), (T4 : 80\%)\}$. In this example, the local testers’ weight deviations for peer $p_j$ can be calculated as follows:

$\left| (T1:50 - T1:40) + (T2:60 - T2:70) + (T3:0 - T3:20) + (T4:80 - T4:0) \right| = \left| 10 - 10 - 20 + 80 \right| = 60$.

Simulations with 50 rounds each are performed for the parameters $f = 0.7$ and $f = 0.9$. Here we use $p = 5\%$. In Figure 4.8 we report results for $k = 2$ and $k = 1.1$. The experiment suggests that the best parameter values in these situations are $f = 0.7$ and $k = 2$.

$$GlobalDeviation = \sum_{j=0}^{n-1} \left| L_U(p_j)_{\text{time}(k)} - L_U(p_j)_{\text{time}(k-1)} \right| . \quad (4.8)$$
4.5.5 Casualties

For this experiment, we introduced 20 malicious testers (controlled by an attacker). We want to count the number of peers who eventually end up making decisions relying on these malicious testers. Here, we consider that any peer who has more than half of its Used Testers List \( L_U(p) \) from the introduced malicious testers is an affected peer. This indicates that the majority of its testers are controlled by the attacker. An experiment based on simulations with 50 rounds each, for the parameter \( f = 0.7 \), is used to detect the best value for the parameters \( p \) and \( k \). The results in Figure 4.9 are averaged over 10 instances for each of the values 5\%, 15\%, and 25\% for the parameter \( p \) and the values 1.1\%, 2\%, and 3\% for the parameter \( k \). The experiment suggests that the lowest number of affected peers can be achieved with low values for parameters \( p \) and \( k \) in this situation is \( p = 5\% \) and \( k = 1.1 \) as shown in 4.9.

4.6 Conclusions

We design and propose a distributed recommender system for advertising testers based on heuristics of proximity and diversity meant to improve the chances of independence
of the used testers. The independence of the used testers is essential for the resistance to Stacking the Deck Attacks SDA (where the attacker can orchestrate taking over the control of the update process).

A set of metrics is defined for quantifying the promise of the investigated distributed recommendation system. These are: Distribution, Local Stability, Global Stability, and Casualty Rate. We use simulations to evaluate the parameters of the proposed mechanisms for distributed recommendation of update testers, looking for the most promising results with respect to the aforementioned metrics.
Chapter 5

Supernode Services based on Incentives

5.1 Motivation

In Chapter 4, we proposed a novel mechanism for suggesting testers to end-users via a distributed P2P recommendation system. Accordingly, in this chapter we focus on the communication level between peers in a P2P network. Our contribution here is to use a set of incentives for reducing free-riding effects in open-source P2P systems. In this chapter, we introduce a new fully decentralized unstructured peer-to-peer (P2P) approach to open-source instant messaging systems.

5.2 Introduction

Instant messaging has been one of the most successful applications on the Internet, from talk and IRC to Twitter, WhatsApp, and Facebook. One of the most currently used such systems, Skype, is a peer-to-peer (P2P) application, which has more than 500 million user accounts and has more than 50 million active users per
day [73]. The P2P architecture offers Skype significant robustness and scalability, as well as efficiency, particularly from the perspective of a low latency. At its beginning Skype held costs down by using the machines of its users as supernodes. Since it was a closed software, its users could not disable the supernode function before the release of Version 3.0 [155, 118]. The possibility to disable this function was offered due to demand from enterprises [45, 119, 62] and came in parallel with the development of a (potentially costly) Skype-owned infrastructure. Disabling the supernode function requires advanced users to deal with the Windows registry [155]. Moreover, based on our survey, most users are not even aware of the existence of this option.

While efforts to build open-source Skype clients have failed due to secret changes in protocols [10, 11, 51], it is questionable whether Skype could in fact have survived its early years as a decentralized service if the protocol would have been open. The issue is that, with open-source software, the users can easily get versions that disable expensive supernode functions.

System functions from which users have incentives to not deviate are said to have the faithfulness property [137]. In the closed-source chat systems, the only incentive to act as a supernode is the fact that either binaries do not allow users to disable that function selectively or users are unaware of it. Since current protocols have no intrinsic incentives for voluntarily acting as a supernode, availability of an open-source agent can starve the network of super-nodes and lead to the demise of the service [73]. Based on our survey of frequent users of Skype, 87% of the users will disable the supernode function as long as they still can have the same benefits.

In this report, we introduce a set of incentives that can motivate peers to run the function of supernode on their personal devices or on dedicated machines. For example, peer Bob can use his desktop computer as a supernode to help his friends (who are
behind NATs/firewall or who are travelling and thereby changing their physical IP addresses) to connect with other peers. Peer Bob, on average, is available for a limited time (e.g., 4 hours/day). On the other hand, an advertising agency Trent may want to spread advertising messages through the network, reaching as many users as possible. One idea it can exploit is to have multiple dedicated servers to run as supernodes (or directory servers), specifying as their terms of usage the acceptance of advertisements. Clients who want to register to (or inquire) these directories should accept such terms. Just like company Trent, any other companies, organizations, groups, or individuals can have their own dedicated supernodes (directory-servers) based on the types of incentives relevant to them. To encourage peers to support the communication of others, the proposed protocol enables incentives, such as:

- Ability to control the traffic passing by her system:
  - helping the endpoints of the communication (e.g., based on friendship or subject of communication).
  - rejecting the endpoints of the communication (e.g., based on disagreement).

- Getting information concerning who talks to whom and what are they talking about (for research, marketing data, or pure curiosity) [98, 127].

- Supporting noble causes such as availability of open-source P2P chat systems, thereby supporting the freedom of customization.

- Opportunity to insert advertisements into the stream.

- Ability to offer paid services of communication support.
This variety of alternatives for the ownership of the dedicated-supernodes can increase the *decentralization*. Decentralization is desirable in P2P networks since it increases robustness to the economic troubles of a few players. Verifying that the owners of the different supernodes are different is in general a difficult problem, but in our framework, many users can circumvent it by selecting supernodes offered by friends or known entities.

In the next section, System Architecture, we introduce the main concepts and the components of the system. The protocol section describes in detail the mechanism used by peers for incentive chat. After presenting experimental results and used metrics, we end by discussing them in the Conclusion section.

### 5.3 System Architecture

Any entity in the proposed peer-to-peer network can simultaneously be an ordinary node (ON) and a supernode (SN). An entity functions as a supernode only if its user enables
Figure 5.2: **P2P System Architecture** contains two types of entities: supernodes (SNs in blue) and ordinary nodes (ONs in green)

it, and if it has the appropriate resources. This is unlike the Skype overlay network that has an extra entity: the login server [10].

In our approach, the software that represents each peer (agent) consists of four components: Identity Management, Incentive Management, Communication Management, and Supernode Management as illustrated in Figure 5.2. Peers have equal privileges and play the roles of client and server at the same time.

### 5.3.1 Identity Management

Each node in the network has a *logical identifier* and a set of *physical addresses*. Each human user (peer) has a unique global identifier: peer GID. The GID is a public key of the user; therefore, the chance of generating two identical GIDs is extremely low. This mechanism of generating the peer GID removes the need for using a centralized server for registering new users (peers) to guarantee the uniqueness of the username, as in Skype [10]. An example of deployment by a human user (peer) can have multiple installations of the agent software on different devices, such as a desktop, laptop, or cell phone. Each of these installations is referred to as an *instance of the peer* and is
identified using a string (a name that should be unique for that peer). The combination between the GID and the instance-name uniquely identifies each instance, defining the logical identifier of our nodes. A physical address is of the type \((\text{protocol}; \text{IP}; \text{port})\).

Each peer can share his contact information with other peers by using his address container. The concept of the address container is related to the concepts of handle and user-profile in Diaspora [18], extended with a public key transfer to integrate security features. An address container is an object (e.g., a file) that contains information about a peer, such as:

- the name and GID of the peer
- a list of peer instances—each peer has one or more instances (installations)
- the certificate of the peer, usable to verify its identity on connection
- a list of common socket addresses (IP and port) where the peer can usually be contacted (set of physical addresses)
- a list of directory servers supporting this peer with connections when he is not at one of the common socket addresses

### 5.3.2 Incentive Management

Our study assumes rational behavior, where users of different types can have utilities that depend on various incentives. The incentives we address are categorized into the following types:

1. **Curiosity**: e.g., getting information about who talks about what to whom
Figure 5.3: **Term Agreement**: Negotiation Between Supernodes and Ordinary Nodes

2. **Commercial perspective**: e.g., opportunity for paid services or inserting advertisements

3. **Altruism**: e.g., supporting unknown or known others to freely communicate

Each supernode provides communication services to other nodes based on a set of agreed terms, as illustrated in Figure 5.3. The parameter *terms* sent with negotiation messages consists of a structure of the type $OR(term_1, term_2, ...)$, i.e., a disjunction of choices, where each $term_i$ is a pair of the form $(type, requests)$ where *type* specifies the communication mechanism possible with the requested peer (direct, forwarded, staring up, or registering with the supernode), while *requests* is a tuple $\langle topic, plaintext, ads, payment \rangle$, each of these elements specifying whether the corresponding incentive is requested and how.

### 5.3.3 Supernode Management

A supernode is an agent that is not found behind some kind of firewall, is publicly addressable, and provides communication services to other peers.
Each peer with sufficient resources can optionally play the role of supernode by acting as directory for her registered peers or by forwarding incoming messages to peers behind NATs or firewalls. Each supernode has three functions:

- **Directory Services**: Supernodes act as directory servers and provide the physical addresses of their registered peers when queried. As in Figure 5.4, to contact P1, P2 requests help from directories that have the physical addresses of P1.

- **Forwarding Messages**: Supernodes relay STUN signals or other communications between two other nodes, both of which may be behind some kind of NATs or firewalls.

- **Peer Registration**: Supernodes provide communication services for their registered peers. Peers can subscribe with one or more supernodes. As in Figure 5.4, P1 subscribes with several directories of her choice.

### 5.3.4 Communication Management

A **NAT** is a mechanism to share a network connection with a unique IP from several machines on an intranet. Each socket on a machine on the intranet is viewed from
the outside as having the same IP, but potentially a different port than the one actually used on its own machine. Some NAT devices support automatic port forwarding configuration (allocating a public Internet socket for any server of the agent), and then the agent can be considered to be publicly addressable, and the NAT is no longer relevant for the communication. Otherwise, the protocol has to embed STUN, and therefore we now introduce NATs and our version for the STUN protocol.

There are several types of NATs. When both peers are found behind NATs and one of the NATs is symmetric, then all the communication has to be performed using an external server (e.g., with the TURN protocol). With full-cone NATs, two devices both found behind such NATs can communicate directly if their communication is initialized by an external server (e.g., with the STUN protocol).

For Full-Cone NATs, once the communication is launched, the supernode can be bypassed as soon as the peers can exchange their Internet addresses. In this case the incentive that a supernode can get consists of its ability to learn that users do communicate. To reduce the chance that it provides support to a cause it opposes, or is used just for gathering data, the supernode can request a declared topic for each connection request. For example, if the declared topic matches items on a black-list, the requesting user can be denied. This can be used to require the peers to study advertisements.

With Symmetric NATs, all the communication has to pass via a relaying supernode (Figure 5.1). The load of this supernode is significant since it has to transmit the whole data. An incentive automatically available to a supernode based on open-source is that it can learn the amount of communication between peers, knowledge that can provide motivation to some researchers, marketing departments, etc. Other supernodes put other conditions (other than payments):
• being allowed to insert ads in the stream of data. While peers may insert
digital signatures on blocks of data to be able to automatically remove such ads,
supernodes may request dynamic interaction to verify that peers have read the ads
(such as CAPTCHA: ask receivers to answer a question based on the ads they
were supposed to see).

• being given access to communication in plaintext (a request that data not
be encrypted). While some people may still exchange secret data by using
steganography, supernodes can use public data in the communication for
marketing, studies, curiosity, etc.

5.4 Protocols

In this section, we introduce the protocol followed by each participant. First we give the
perspective of the user. A user follows the sequence of interactions below:

1. Download **container** of address from destination’s website/email into her agent.
   This has to be done only once in the lifetime of the relation between the two peers.

2. For an existing peer, try direct connection addresses first, if available. If they
   work, go to step 5.

3. Send **request** message (**Help**) to a peer’s directories, in parallel. Available
directories answer with **negotiation** messages (**Welcome**).

4. Choose the directory $S$ with the best terms and send an **agree** message
   (**Confirmation**).

5. Send **data** message based on the obtained procedure.
In the scheme above, at the first step the agent saves the address in the local storage for potential further usage.

The protocol of a supernode is given in Algorithm 9. The supernode uses as data structure a hashtable \textit{channel} with peer GIDs and logical identifiers as keys and addresses as values. Supernodes handle the \texttt{request} messages with the procedure \textit{ForwardRequest}. The \texttt{agree} messages are handled with the procedure \textit{AgreeForwarding}, and the \texttt{data} messages are handled with the procedure \textit{ForwardingData}. Note that these procedures enable communication behind symmetric NATs, but this is sufficient to start any other type of communication possible, once peers can communicate with each other and if their communication is not censored. Nevertheless, for direct communication from behind two full-cone NATs, they additionally need the cooperation of the supernode in providing them with their external addresses.

In order to understand the communication initialization between peers, we illustrate four cases that could appear when a peer tries to connect to another peer (which mirrors related cases of the STUN protocol, enriched with handling of incentives). For clarity, we first present the four cases with a potentially inefficient version of the initialization, which increases the latency by one round-trip. Subsequently, we show that the latency of this initialization can be reduced in certain circumstances to the latency for the case without incentives (STUN):

\textbf{Case Roaming Peer (No NAT).} In this case, peer\_A wants to send a message to peer\_C, but peer\_A does not have the current address of peer\_C (see Figure 5.5). On the other hand, peer\_B has peer\_C’s address stored in its database (peer\_B acts as a directory...
Algorithm 9: Supernode Protocol

```
when ForwardRequest(src_IP, source, destination)
  authenticate(src_IP, source);
  if not willingly_forwarding(source, destination) then
    return;
  terms → myterms[source];
  send negotiation(terms) to src_IP;

when AgreeForwarding(src_IP, source, destination, terms)
  check terms;
  if NAT.opening(terms) then
    send (src_IP) to destination;
  else
    open channel[destination] adding destination to set of its tags;
    open channel[source] adding source to set of its tags;

when (ForwardingData(source, destination, message))
  if (!check_forwarding(source, destination) then
    return
  send data message to channel[destination];
```
server for peer C). Therefore, peer A needs peer B’s help in order to communicate with peer C. Both peer B and peer C have public IP addresses.

1. Peer A sends a help request to peer B. The request has two parameters (1) destination information. In this case, it is peer C public identification (PID) and optionally (2) topic information (why does peer A want to communicate with peer C).

2. Then, peer B sends a welcome message to peer A including peer B’s terms to help. As aforementioned, the terms include:

(a) incentives: access to plaintext messages, insertion of advertisements, etc.

(b) services: forward only (address of peer is not offered due to symmetric NAT or to preclude bypassing), peer current address, establishing a connection (for full-cone NAT).

3. Peer A sends an acceptance confirmation message to peer B.

4. Peer B sends a message to peer A containing the contact information of peer C (IP address and port).

5. Finally, peer A can connect to peer C.
Figure 5.5: [Case 1] Peer_A can reach peer_B but does not know the current IP of peer C. Peer_B acts as a directory server for peer_C.

**Case Full-Cone NATs.** In this case, both peer_A and C are behind full-cone NATs (see Figure 5.6). They can reach peer_B but they cannot reach each other. Peer_B has a communication channel with peer_C. Therefore, peer_A needs peer_B’s help in order to communicate with Peer_C. The solution here is based on hole punching for UDP or TCP [43].

1. Peer_A sends a help request to peer_B.

2. Then, peer_B sends a welcome message to peer_A including peer_B’s terms for help.

3. Peer_A sends an acceptance confirmation message to peer_B.

4. Peer_B sends a contact message to peer_C containing the contact request information of peer_A (IP address and port).

5. Peer_B sends an address message to peer_A containing the contact information of peer_C (IP address and port).

6. Peer_A and peer_C pierce their NATs.
7. Finally, peer_A and peer_C have a communication channel to exchange messages.

![Figure 5.6: Case 2](image)

Figure 5.6: [Case 2] Peer_A and B are behind full-cone NATs. They can reach peer_B but they cannot reach each other.

**Case Symmetric NAT.** In case both peer_A and peer_C are behind a NAT and one of the two NATs is symmetric (pictured at peer_C in Figure 5.7), then forwarding between them cannot be avoided.

1. Peer_A sends the help request to peer_B.
2. Peer_B sends a welcome message to peer_A including peer_B’s terms to help.
3. Peer_A sends an acceptance confirmation message to peer_B.
4. Peer_A sends a message to peer_B containing a message.
5. Peer_B forwards the message to peer_C with the data of the sender and potentially with additional processing (insertion of advertisements, extra negotiation with peer_C).
6. Answers from peer_C follow the same scheme.
Figure 5.7: [Case 3] Peer_A and C are behind NATs, at least one of the NATs being symmetric. They can reach peer_B but they cannot reach each other.

**Case NAT at Destination.** The case when the initiator peer_A is not behind a NAT but peer_C is in this situation and has a communication channel open with peer_B, then the procedure is similar to the first case (see Figure 5.8).

1. Peer_A sends a help request to peer_B.
2. Peer_B sends a welcome message to peer_A including peer_B’s terms to help.
3. Peer_A sends an acceptance confirmation message to peer_B.
4. Peer_B sends a message to peer_C containing the contact information of peer_A (IP address and port).
5. Finally, peer_C can connect to peer_A.
Figure 5.8: [Case 4] Peer_C is behind the NAT. Peer A can reach peer_B but cannot reach peer_C.

Reducing Latency of Initialization. The previously described simplified handshake mechanism has a higher latency than STUN due to the extra round-trip used to agree on the terms for service.

This handshake can be brought to the latency of STUN whenever the client accepts to proactively offer the services that the supernode requests. The Help message is allowed to optionally contain the list of terms that the client is ready to accept (willing to do list). Instead of exchanging three messages between peers for a term agreement—Help, Welcome and Confirmation—the client only needs to send a single message: Help. In case the supernode finds acceptable terms among the terms offered by the client, then it returns the most preferred among them, term to peer_A using an Accept message. The example in Figure 5.9 details the scenario of full-cone NATs, but it applies identically to the other discussed cases. If the list of terms does not match its acceptable terms, then it can reply with a RejectRequest message specifying its terms. The client can eventually retry the communication request including some of the requested terms.
The peer is configured by a user with a list of acceptable terms tagged with a preference value: \( \langle (term_1, p_1), (term_2, p_2), \ldots \rangle \). First the software agent may try to negotiate the terms \( term_1 \) with the highest preference \( p_1 \) using known supernodes serving peer C. If that fails, then it can try \( terms_2 \) with the lower preference value \( p_2 \), until it exhausts its acceptable terms. The user is presented with the terms suggested by supernodes in their rejection replies. Users can change their default terms for certain supernodes, based on answers to their past requests, to reduce the latency of subsequent connections. The agent can automatically infer based on recent past answers whether a query will fail and optimize the query of supernodes.

**Addressed attacks**

- Fake service claim: The expected behavior of a supernode is to provide communication services to registered peers based on a settled term agreement. However, some supernodes may take advantage of this function in different ways. Some attacker may claim to provide the supernode function (forwarding messages, providing addresses or startup communication) just for gathering start-up data, without actually providing services. It is up to the peer using it as a directory to detect and protect herself from these attacks, and a recommendation system can be built to evaluate supernodes.

Figure 5.9: Reducing latency
• False identity declarations: One can use protocols such as webfinger [110]. Intuitively, some identities are verified directly using a protocol based on SMS and previously-known email addresses.

• Supernode evasion: a user could indeed ask the address of an intermediary as a way of hiding the final destination. Peers can use steganography for exchanging text privately over a supervised channel. These techniques are common even now with existing email infrastructures (see anonymous remailers [142]), and we expect that they will happen with our system, too. It already impacts trust in claims of NonGovernmental Organizations (NGOs) as to the real purpose of their activities, but it does not lead to a lack of survivability of NGOs. These potential attacks somewhat reduce the incentives of type one (curiosity), and somewhat increase the incentives of type three (serving selectively certain causes). They have less impact on incentives of type two (commercial perspective).

5.5 Methodology and Experiments

We now describe the methodology we propose for evaluating the impact of given incentives on the survivability of an open-source system. Here we exemplify with the described case study of our instant messaging system. In order to quantify the effects of proposed incentives on the survivability of the proposed approach, we have conducted a randomized survey of $M = 223$ frequent users of Skype. The results of the survey, in conjunction with simulations, are used as a base for our evaluations.
5.5.1 Why Simulations and Survey?

Even though the proposed protocol is implemented in a real system (DDP2P [138]), simulations were needed for experiments since the real system’s intrinsic privacy precluded gathering statistics (and the size of its user population is still small). In order to make our simulations more realistic, we designed an online questionnaire to collect the required data from Skype users. We targeted Skype users in our survey because of the nature of this study which raised a research question; “Would Skype-like systems be viable if their protocol is open and users could easily disable the supernode function?”

5.5.2 Survey Design and Implementation:

The process of the survey consists of four stages:

1) Designing questions: In Appendix B, we show an English version of the questionnaire. In this questionnaire, we designed questions with respect to the following:

- Questions should be easily understood by common users (e.g; no technical or specialized words like supernode, P2P network, bandwidth, ...),

- Biased questions should be avoided. This can be done by using indirect questions (e.g., ”If Skype allows you to select whom to serve, would you allow your friends to use your computer resources to connect to others in the Skype network?” This question is used to study the effect of the friendship-based incentive on the survivability of the system),
Questions should flow logically from more general to more specific and from factual and behavioral questions to attitudinal and opinion questions” (Robson 2002).

Questions in the questionnaire are divided in four categories:

a) questions about participants’ Skype usage

b) questions about participants’ devices information

c) questions about participants’ background regarding ’Skype user agreement’

d) questions about participants’ preferred incentives

2) Pretest of the survey: In order to obtain the intended information, we pretested a smaller subset of the target sample to ensure the clarity and the accuracy of the designed questionnaire. Therefore, we used a face-to-face interview of a sample of 30 Skype users. In this stage, we collected the feedback of the respondents about the questionnaire design and the clarity of the questions in addition to the questionnaire answers. This helped us to refine and improve the questionnaire before distributing it online.

3) Final survey distribution: The survey was implemented as a web questionnaire in two languages (English and Arabic) to reach a wider sample. We posted the questionnaire online from 23.5.2014 to 18.8.2014. In order to advertise the questionnaire, we used social-media (Twitter, WhatsApp, and Facebook) to distribute reference links to our survey. In order to limit and reduce the sample-selection bias, we used different identifiers (usernames) in the used social-media.
4) **Data collection and analysis**: after two months of posting the survey, we collected data of 223 respondents and we used this data to feed our simulations and then produce results (Figure 5.10).

![Figure 5.10: Integrating Survey and Simulations](image)

**5.5.3 Survey Statistics**

After gathering and analyzing the feedback from the subject participants, we summarized the asked questions and obtained statistics of our sample, shown in Table 5.1. The comparative chart for the popularity of incentives is shown in Figure 5.11.
<table>
<thead>
<tr>
<th>Statistics</th>
<th>Indications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skype users who will disable the supernode function as long as they still can have the same benefits.</td>
<td>87% Free-ride behavior</td>
</tr>
<tr>
<td>Percentage of users who are willing to enable their supernode function based on the causes that they support.</td>
<td>39% Enabling SN function</td>
</tr>
<tr>
<td>Percentage of users who are willing to enable their supernode function based on helping friends (friendship).</td>
<td>69% Enabling SN function</td>
</tr>
<tr>
<td>Percentage of users who are willing to enable their supernode function if they get paid for the provided services.</td>
<td>40% Enabling SN function</td>
</tr>
<tr>
<td>Percentage of users who are willing to enable their supernode function based on revealing the exchanged messages.</td>
<td>29% Enabling SN function</td>
</tr>
<tr>
<td>Percentage of users who are willing to enable their supernode function for free.</td>
<td>13% Enabling SN function</td>
</tr>
<tr>
<td>Percentage of users who will not enable their supernode function for any reason.</td>
<td>14% Not enough incentives</td>
</tr>
<tr>
<td>Percentage of users who are willing to reveal the communication case/topic to get communication services from SNs.</td>
<td>11% Offering non-monetary incentive</td>
</tr>
<tr>
<td>Percentage of users who are seeking help from friends to get communication services from SNs.</td>
<td>55% Offering non-monetary incentive</td>
</tr>
<tr>
<td>Percentage of users who are willing to pay a certain amount of money to get communication services from SNs.</td>
<td>27% Offering monetary incentive</td>
</tr>
<tr>
<td>Percentage of users who are willing to receive advertisements from SNs for communication services.</td>
<td>39% Offering monetary incentive</td>
</tr>
<tr>
<td>Percentage of users who are willing to send plaintext (unencrypted messages) through SNs to get communication services from these SNs.</td>
<td>9% Offering non-monetary incentive</td>
</tr>
</tbody>
</table>

Table 5.1: Survey Statistics

132
Figure 5.11: **Comparing Incentives:** Based on the survey, the chart shows the percentage of the agreement on each incentive from two perspectives: 1) Incentives accepted by prospective supernode owners for enabling the supernode function (blue) and 2) Incentives offered to get the necessary communication services from supernodes, with two types of incentives: a) Monetary Incentives (green), including direct money payment (light green) or payment by advertising (dark green) and b) Non-monetary incentives (brown), including declaring a subject of the communication, exchanging unencrypted messages, and getting help based on friendship.

### 5.5.4 System Simulation

Assuming users behave as per our survey feedback, we simulated the overlay peer-to-peer network of our incentives-based-chat system. We used this simulation in order to evaluate the survivability of the system. Each participant in the survey is replicated by a set of peers in the network. Each instance of a peer (node) in the simulation is characterized by the attributes in Table 5.2.
<table>
<thead>
<tr>
<th>Property</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance</td>
<td>The device name (e.g., desktop1)</td>
</tr>
<tr>
<td>number_of_instances</td>
<td>The number of devices owned by the participant</td>
</tr>
<tr>
<td>link_to</td>
<td>Avg. number of contacts (neighbor peers)</td>
</tr>
<tr>
<td>active_contacts</td>
<td>Regular contacts</td>
</tr>
<tr>
<td>candidate_SN</td>
<td>Supernode quality level (Supernode Candidacy)</td>
</tr>
<tr>
<td>availability</td>
<td>Avg. connection time to the Internet (e.g., 3 hours/day)</td>
</tr>
<tr>
<td>type</td>
<td>Device type (desktop, laptop, tablet, and cellphone)</td>
</tr>
<tr>
<td>terms_to_help_as_SN</td>
<td>List of the required incentives to be satisfied to serve other peers</td>
</tr>
<tr>
<td>willing_to_give</td>
<td>List of offered incentives to get help from the registered SNs</td>
</tr>
<tr>
<td>is_behind_NAT</td>
<td>(true/false) based on probability p=0.6 [144]</td>
</tr>
<tr>
<td>registered_with</td>
<td>List of registered supernodes that keep current instance address</td>
</tr>
<tr>
<td>covered_time</td>
<td>number of hours that a peer instance can be covered by its list of registered supernodes during the day (24h)</td>
</tr>
</tbody>
</table>

Table 5.2: Peer Instance Properties

The following shows the setup of our P2P network:

- **N** peers \((P_1 .. P_N)\): each participant \(i\) out of \(M\) participants in the survey is replicated by the set of peers \(\{P_j | \exists k \in \mathbb{N}, j = i + k * M, j \leq N\}\).
A subset of $S$ instances of these peers voluntarily acts as supernodes based on the answers from the survey related to: 1) number of instances in the system. 2) peer-instance willingness to be a supernode.

Each peer $P_i$ is linked to $n_{i \mod M}+1$ other peers, where $n_{i \mod M}+1$ is the answer of the survey question regarding the number of contacts the participant has in his messaging system.

Each peer $P_i$ has $m_{i \mod M}+1$ instances, where $m_{i \mod M}+1$ is the number of devices declared by the corresponding survey participant.

Each peer $P_i$ is registered with $k_i$ supernodes, based on need and test parameters.

**Supernode Candidacy**

Any regular node with a public IP address having sufficient CPU, memory, uptime, and network bandwidth is a candidate to become a supernode [10, 156]. We introduce a quantitative measure of the degree to which a node is desirable to function as a supernode, and we call this measure: *candidacy level*. This candidacy level varies from one node to another. For example, a powerful desktop connected to the Internet 12 hours/day has a different supernode candidacy level than a slower device (e.g., common laptop) connected to the Internet 2 hours/day. We summarized the supernode candidacy level to be based on three variables:

1) Type (Power): It has four possible values: desktop, laptop, tablet, and cellphone (while we know that cellphones can be built to have more computation power than some desktops, we assume that in general these values correspond to a smaller computational power). We use the type of the peer device as an indication of the expected power and capacity of the node (CPU and memory).
2) Availability: the time that a peer is connected to the Internet (with assumption of enough network bandwidth). Time here is a continuous variable; therefore, we use four categories based on the number of hours a node is connected to the Internet. This variable has four possible values: 1-6 hours, 7-12 hours, 13-18 hours, and 19-24 hours. The chart in Figure 5.12 shows the supernodes’ availability during the day based on our simulation.

![SN Availability In 24 Hours](image)

Figure 5.12: **SN Availability During the Day**: The chart shows the percentage of the available SN (out of all SNs in the system) for each hour during the day. For example, the average availability of the SNs from [12am-6am] is 33.2% while the average availability of SNs in the evening [6pm-12am] is 66.2%

3) Behind NAT: nodes that are not behind NATs or firewalls (or use port forwarding) can be easy to reach from other nodes; therefore, they are good candidates to be supernodes (they have public IPs). It is reported [144] that 60% of P2P users are behind some kind of NAT. There are two values for this variable: true or false.

We use a simple Bayesian Network as a way to represent the probabilistic relationships between the variables: type (t), availability (a) and behind NAT (b), and the candidacy level of a supernode(s) (see Figure 5.13).
Figure 5.13: **Bayesian Network:** The degree of supernode candidacy is influenced by: device type, peer-instance availability on the Internet, and peer-instance location if behind some kind of NAT or firewall.

In Table 5.3 we illustrate a sample Conditional Probability Table (CPT) to demonstrate the probability of the candidacy level of a supernode(s) with respect to the other three variables. The table has 32 rows (4*4*2).

\[
P(s \mid t, a, b)
\]

For example, a desktop computer that has a public IP (not behind a NAT or firewall) running and connecting to the Internet for five hours, has a high probability to be a reliable supernode. We can represent this by \( P(s \mid "desktop", "1–6", false) = 0.85 \). On the other hand, the probability of being a reliable supernode for a desktop available for five hours but running behind some kind of NAT or firewall (potentially with port forwarding and firewall exceptions) is smaller. This can be represented by \( P(s \mid "desktop", "1–6", true) = 0.35 \). The numerical data was extracted from surveys.
The measures used to evaluate the impact of each incentive on the survivability of the system are:

- **Number of peers that find no supernode.** This measure shows how many peers find no supernode serving for the incentives that the peers are willing to offer.

- **Coverage of least served peer.** This measure shows the number of hours of coverage for the peer with the smallest number of covered hours (hours when it is served by a supernode).

- **Average percentage of covered time.** This measure shows the average number of hours that a peer is served by its supernodes.

- **Percentage of peers covered 24h.** This measure shows how many peers have at least one supernode serving them at any time.
• **Maximum number of needed supernodes.** This measure shows the largest number of supernodes that some peer had to register with, in order to achieve its maximum coverage.

• **Average number of needed supernodes.** This measure shows the average number of supernodes that peers had to register with, in order to achieve their maximum coverage.

**Experiments**

We ran experiments with M=223 and N=10000 as the following:

• **Unique Incentive.** In this experiment we compared the discussed incentives based on the measures obtained when only one type of incentive is enabled (besides the supernodes that serve liberally). The results for the discussed incentives are shown in Table 5.4, showing the significace of supporting friendship and advertisements.

<table>
<thead>
<tr>
<th>Consequences</th>
<th>Using all incentives</th>
<th>Only use monetary incentive</th>
<th>Only use advertisement incentive</th>
<th>Only use friendship incentive</th>
<th>Only use cause incentive</th>
<th>Only use plaintext incentive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. number of SNs a peer registers to</td>
<td>3.6712 (SNs/peer)</td>
<td>2.2101 (SNs/peer)</td>
<td>2.599 (SNs/peer)</td>
<td>3.0832 (SNs/peer)</td>
<td>1.6621 (SNs/peer)</td>
<td>1.6034 (SNs/peer)</td>
</tr>
<tr>
<td>Maximum number of SNs a peer registers to</td>
<td>15 (SNs/peer)</td>
<td>15 (SNs/peer)</td>
<td>14 (SNs/peer)</td>
<td>16 (SNs/peer)</td>
<td>14 (SNs/peer)</td>
<td>14 (SNs/peer)</td>
</tr>
<tr>
<td>Avg. coverage time for each peer</td>
<td>24.0 (hours/day)</td>
<td>13.125 (hours/day)</td>
<td>15.876 (hours/day)</td>
<td>19.5341 (hours/day)</td>
<td>9.1728 (hours/day)</td>
<td>8.9952 (hours/day)</td>
</tr>
<tr>
<td>Coverage of least served peer</td>
<td>24 (hours)</td>
<td>18 (hours)</td>
<td>24 (hours)</td>
<td>6 (hours)</td>
<td>24 (hours)</td>
<td>24 (hours)</td>
</tr>
<tr>
<td>Percentage of peers covered 24 hours</td>
<td>100 %</td>
<td>54.68%</td>
<td>66.15%</td>
<td>80.74%</td>
<td>38.22%</td>
<td>37.48%</td>
</tr>
<tr>
<td>Number of peers who cannot register with any SN</td>
<td>0</td>
<td>4531</td>
<td>3385</td>
<td>1845</td>
<td>6178</td>
<td>6252</td>
</tr>
</tbody>
</table>

Table 5.4: Unique Incentive

• **Absence of Incentives.** In this experiment we compared the impact of the lack of support in the system for a given incentive when all other considered incentives
(except for free service) are supported. This experiment can show the overlap between the populations covered by the given incentives and can help remove redundancy (supporting incentives that make no difference). Our results from this experiment are shown in Table 5.5. This shows that friendship and advertisements are powerful mechanisms to match peers to supernodes. People claim to be more inclined to pay money or spend time on advertisements rather than to explicitly lose some of their privacy.

If one takes into account the offers of free service (from people simply willing to support anybody with no expected benefit), then we find that the removal of any given single incentive support still allows for everybody to be covered for 24 hours with an average number of approximately 4 supernodes and a maximum number of 16 supernodes.

<table>
<thead>
<tr>
<th>Consequences</th>
<th>Using all incentives (excluding free services)</th>
<th>Absence of monetary incentive</th>
<th>Absence of advertisement incentive</th>
<th>Absence of friendship incentive</th>
<th>Absence of cause incentive</th>
<th>Absence of plaintext incentive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. number of SNs a peer registers to</td>
<td>3.4968 (SNs/peer)</td>
<td>3.046 (SNs/peer)</td>
<td>2.8121 (SNs/peer)</td>
<td>2.4568 (SNs/peer)</td>
<td>3.3803 (SNs/peer)</td>
<td>3.3913 (SNs/peer)</td>
</tr>
<tr>
<td>Maximum number of SNs a peer registers to</td>
<td>13 (SNs/peer)</td>
<td>12 (SNs/peer)</td>
<td>12 (SNs/peer)</td>
<td>13 (SNs/peer)</td>
<td>13 (SNs/peer)</td>
<td>12 (SNs/peer)</td>
</tr>
<tr>
<td>Avg. coverage time for each peer</td>
<td>23.1234 (hours/day)</td>
<td>20.5346 (hours/day)</td>
<td>18.8203 (hours/day)</td>
<td>16.032 (hours/day)</td>
<td>22.3724 (hours/day)</td>
<td>22.4807 (hours/day)</td>
</tr>
<tr>
<td>Coverage of least served peer</td>
<td>12 (hours)</td>
<td>12 (hours)</td>
<td>15 (hours)</td>
<td>24 (hours)</td>
<td>15 (hours)</td>
<td>18 (hours)</td>
</tr>
<tr>
<td>Percentage of peers covered 24 hours</td>
<td>95.8%</td>
<td>84.86%</td>
<td>77.83%</td>
<td>66.80%</td>
<td>92.65%</td>
<td>93.19%</td>
</tr>
<tr>
<td>Number of peers who cannot register with any SN</td>
<td>358</td>
<td>1435</td>
<td>2151</td>
<td>3320</td>
<td>671</td>
<td>627</td>
</tr>
</tbody>
</table>

Table 5.5: Absence of Incentives: Not Accounting for Free Services

- **Robustness of Incentive Declaration.** In this experiment we evaluated the robustness of our measures to the correctness of the declarations by participants in the survey. Since there is a possibility that some participants did not correctly understand some questions, we evaluated the impact of potential small errors in
the survey result on the measures obtained by simulation. For this purpose, we added/removed some incentives of the given types from the lists of incentives that participants claimed to be willing to offer, and we checked the slope of the curves that this generates for a given measure. For example, since the nature of possible noble causes a participant may be willing to serve is difficult the easily foresee, in Figure 5.14 we compute the impact of variations in declarations of pursued noble causes on the number of peers that find no supernode to serve them. One can therefore observe that the impact is smooth, implying that no big change comes from small errors in the survey. However, the impact of large errors (11% - 41%) changes results by a 2:1 ratio.

![Graph showing the impact of variations in declarations of pursued noble causes on the number of peers who cannot register with any SN based on the percentage of peers who are willing to reveal their causes to supernodes.](image)

Figure 5.14: Only use "revealing cause" as an incentive: The chart shows the number of peers who cannot register with any SN based on the percentage of peers who are willing to reveal their causes to supernodes. This experiment is enabling only revealing cause as an incentive.

### 5.6 Conclusions

We address the problem of providing incentives for users to let their systems serve as supernodes (helpers for initiating or forwarding communication) in an open-source P2P protocol (e.g., chat). We remark that one of the P2P applications commonly used for
chat, Skype, does not have the *faithfulness* property. Namely, the only reason for which certain users allow their Skype agent to relay messages is that they cannot turn it off (given that it is difficult in the close source software).

The research question addressed here is how to provide intrinsic incentives for peers to serve as supernodes, in a way that would make an open-source chat protocol viable. The observation is that human volunteers have several ways in which they can benefit from helping with establishing connections between others. Some benefits come from common characteristics of humans.

A human benefit for volunteering to send data is the good feeling of *serving a noble cause*. While some volunteers would simply offer their resources for the cause of *open-source*, others may want to learn a declared *topic* of the discussion or to even request access to the whole communication (in plaintext), to guarantee that they are happy to support the *cause* of the given connection. This may be acceptable to certain users and for certain communications. Another human characteristic that can motivate a volunteer is *curiosity* of who talks to whom, of how much they talk and, if public, on what topics (data frequently gathered by marketing departments). Closed software can always obtain this data from people, but in our case users themselves gain by knowing how much information they actually give away.

A third intrinsic benefit of the supernodes can be commercial. The collected data can be used for marketing, studies, etc. Supernode users can also get credit for a *quid pro quo* help if in the future they will be travelling or they lack a publicly addressable IP. Moreover, in some models (e.g., with symmetric NATs or a STUN setup), the supernodes can be enabled to insert advertisements into the stream of data. Just as in the previous cases, the same information is currently leaked to closed source software, while the advantage of open-source is to let the users know exactly what privacy they
are losing. Based on volunteer supernodes, we now establish an incentivized fully
decentralized open-source system for chat.

A significant contribution of this study consists of a methodology to quantify and
compare the power of different incentives. Based on our study and methodology, a
designer of a P2P system can assign priorities to the incentives that he wants to support,
such that he can efficiently assure the faithfulness and survivability of an open P2P
service. The methodology is based on surveys and simulations integrating the results
of these surveys into a model of the system. We also provided details on a case study
implementation for a chat protocol exploiting the studied incentives. Our experiments
for this case show that the incentive with the most impact is friendship, followed by
the support of benefits via advertisements. Other direct monetary incentives come far
behind, trailed by the loss of privacy via revelation of plaintext and causes/topic.
Chapter 6

Conclusions and Discussions

6.1 Summary

We address the problem of the free open source agent software in applications of strategic importance (like petition drives). We address concerns that such software can be the target of Stacking the Deck Attacks from well funded attackers. This attack consists of orchestrating the transfer of the leadership of the development team to people whom they control, enabling a subsequent degradation of the software via automatic updates.

The proposed framework introduces a decentralized authority made up of a cloud of independent testers. Each of these testers can have its own base of users that trust her based on various reasons: reputation, personal contact, or based on independent commercial contracts and services.

Each given user can trust multiple testers with various degrees of trust and can flexibly specify required constellations of Quality of Tests and Results of Tests from these testers in order to automatically accept an update. A threshold trust of any $t_u$ out
of user $u$’s $n_u$ selected testers is just a special case of the possibilities enabled by the proposed framework.

We design and propose a distributed recommender system for advertising testers based on heuristics of proximity and diversification meant to improve the chances of independence of the used testers. A set of metrics is defined for quantifying the promise of investigated distributed recommendation system. These are: Dispersion, Local Stability, Global Stability, and Casualty Rate. We use simulations to evaluate the parameters of the proposed mechanisms for distributed recommendation of update testers, looking for the most promising results with respect to the aforementioned metrics. This recommendation mechanism was validated by a full implementation in DirectDemocracyP2P system.

Based on the nature of our technique for resisting SDA attack in open-source software and the proposed mechanism for suggesting testers to end-users via a distributed P2P recommendation system, we introduce a novel approach for supporting communications between peers in open-source P2P systems. We address the problem of providing incentives for users to let their systems serve as supernodes (helpers for initiating or forwarding communication) in an open-source P2P protocol (e.g., chat). We remark that one of the P2P applications commonly used for chat, Skype, does not have the faithfulness property. Namely, the only reason for which certain users allow their Skype agent to relay messages is that they cannot turn it off (given that it is difficult in the close source software).

Based on our study and methodology, a designer of a P2P system can assign priorities to the incentives that he wants to support, such that he can efficiently assure the faithfulness and survivability of an open P2P service. The methodology is based on surveys and simulations integrating the results of these surveys into a model of
the system. We also provided details on a case study implementation for a chat protocol exploiting the studied incentives. Our experiments for this case show that the incentive with the most impact is friendship, followed by the support of benefits via advertisements. Other direct monetary incentives come far behind, trailed by the loss of privacy via revelation of plaintext and causes/topic.

6.2 Limitations and Future Work

- In Chapter 3, in Section 3.3, we discuss the cases in which attackers can perform certain feats as listed in the threat model. The proposed framework in this research is limited to those attacks, while there exist others that need more research. For example, an attacker can distribute a compromised compiler that can be used by testers to inject malicious code in the produced binaries (updates) [148]. This kind of attack is not new. In 2009 a PhD dissertation by David A. Wheeler: "Fully Countering Trusting Trust through Diverse Double-Compiling” explained how to counter the "trusting trust” attack by using the "Diverse Double-Compiling” (DDC) technique. The same applies to compromised OS or hardware.

- In Chapter 4, we focus on recommending testers based on rating value (weight) while we ignore the diversity of testing qualities. We do not want a user to adopt only testers who only test the same range of qualities (e.g., UNIX compatibility) while ignoring other testers who may have tested a different set of qualities. This can be solved by adding a second recommendation process for each quality.

- In Chapter 5, we introduce a novel approach based on a set of incentives to support communications between peers in open-source P2P systems. The expected
behavior of a *supernode* is to provide communication services to registered peers based on a settled term agreement. However, some supernodes may take advantage of this function in different ways. Some attackers may claim to provide the supernode function (forwarding messages, proving addresses, or startup communication) just for gathering start-up data, without actually providing services. It is up to the peer using it as a directory to detect and protect herself from these attacks, and a recommendation system can be built to evaluate supernodes.
Bibliography


[67] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and
taxonomy of approaches for mining software repositories in the context of
software evolution. *Journal of Software Maintenance and Evolution: Research


[70] Jae Kyeong Kim, Hyea Kyeong Kim, and Yoon Ho Cho. A user-oriented
contents recommendation system in peer-to-peer architecture. *Expert Systems

In *Proc. 5th Int. Conf. on Mobile Computing and Ubiquitous Networking (ICMU2010)*, 2010.

[72] Joseph A. Konstan, Bradley N. Miller, David Maltz, Jonathan L. Herlocker,
Lee R. Gordon, and John Riedl. Grouplens: Applying collaborative filtering


[147] the PEAK developers’ center. Easyinstall.


Appendix A

DirectDemocracyP2P System
DirectDemocracyP2P
— Decentralized Deliberative Petition Drives —

Marius C. Silaghi¹, Khalid Alhamed¹, Osamah Dhannoon¹, Song Qin¹, Rahul Vishen¹, Ryan Knowles¹
Ihsan Hussien¹, Yi Yang¹, Toshihiro Matsui², Makoto Yokoo³, Katsutoshi Hirayama⁴

DirectDemocracyP2P is an open source platform developed in JAVA and offering peer-to-peer and mobile ad hoc wireless communication capabilities. The platform offers an API supporting plugins, beside its main application: deliberative petition drives (aka citizens’ initiatives with integrated argumentation) [1]. An authentication-by-reputation technique based on digital signatures and peer review [2], [3] is integrated into the platform via this main application. Each peer manages independently its database of items of interest. The items of interest are encapsulated as self-contained pieces of information and uniquely identifiable using a system of global identifiers (GIDs). Each GID consists of a combination of public keys with creation dates, or digest values. Communication is based on a combination of push and pull mechanisms. [1]

In the main application, deliberations develop around motions (concept from Robert’s rules of order, similar to petition and initiative), where justifications represent the arguments for gathered signatures (aka votes). Each motion is relevant to an organization (organization = rules defining a constituency and its jurisdiction). Organizations are independent of each other.

The main predefined types of exchanged items are: peer, organization, constituent, motion, neighborhood, witness, justification, vote, news, translation, and tester. These items are synchronized between the peers specifying an interest for them. Users have various possibilities to control what kind of data their agent disseminates and what data it stores. For example, they can block all data related to a given organization, motion or constituent. Alternatively peers may accept only data related to specified organizations, motions or constituents. News items are related to contexts defined by organizations, motions, and justifications. They can be controlled with quotas per constituent and function of their creation date. Translation items refer to the labels of the platform for the graphical interface related to specific organizations. They constitute the basis of a P2P recommendation system for the translations of these labels into the preferred language of the user.

The system also handles plugin-data items that do not have GIDs and are exchanged only between peers specified by the corresponding plugins. Currently there are two distributed applications supported by plugins and available with the system: a spacecraft racing game, and a chat application.

The DirectDemocracyP2P system consists of several modules, such as: Data Handling, GUI, Internet and NAT, Supernode, Ad hoc Wireless, Authentication Census, Automatic Updates, Tester Recommendation, Plugin Interface (Fig. 1).

- The Data Handling module is an interface with the personal database of the peer, saving and reading items with verification of integrity (digital signatures).
- The GUI module allows the user to view and modify the data in her database and the current state of the system.
- The Internet and NAT modules pack and unpack messages to be exchanged between peers via the Internet and contain client and server parts that can communicate both over TCP and UDP (from behind NATs).
- The Supernode module enables users with sufficient resources to support other peers in communicating from behind NATs. It can negotiate incentives.
- The Ad hoc Wireless module is in charge of configuring ad hoc networks and in packing and unpacking messages broadcast over them. It contains a server and a client.
- The Authentication Census module is a reputation system for detecting false identities and Sybil attacks on the collaborative filtering employed by the argumentation.
- The Automatic Updates module monitors a set of user configurable mirrors to download and install software updates that pass a user configurable set of criteria, based on reviews from her preferred testers.
- The Tester Recommendation module is a P2P recommendation system for suggesting testers to users.
- The Plugin Interface module loads available plugin modules dynamically and coordinates the transmission of messages between plugins installed on different peers via the communication system employed by the main application. It also provides a mechanism to exchange certain data (e.g., identity, GIDs) between the main application and plugins, as well as access to local storage and GUI space for menus and applets under the control of plugins.

Fig. 1. DirectDemocracyP2P peer architecture.
I. JOINING OTHER PEERS VIA ADDRESS CONTAINERS

Independent networks of peers can coexist. When user Alice, wants to accept communication from Bob, she exports her address to a file (address container). If the selected file name provided by Alice stands for an existing image, then the address is embedded inside this image. This file/image can be emailed to Bob, or can be posted on the web.

Bob only has to drag and drop the address container to his software agent. If Alice is found behind a NAT, or may move to different IP addresses, she can register herself with a supernode (a node with a stable Internet IP). Each agent has an integrated supernode that an user can enable if she has access to a server with the right resources. The addresses of adopted supernodes are embedded in the address container. An address container encapsulates the GID of the peer (public key), the IP addresses of its machine, and the IP addresses of the supernodes that serve this peer.

II. AUTOMATIC UPDATE

Our solution to security of automatic updates from stacking the deck attacks [5] (i.e., take-over of the development process by powerful players unhappy with a democratic process), inserts independent intermediary testers (software reviewers) between the developers and the end-users. The independent testers can build and test an existing source code revision from an open source repository, and then distribute a signed binary release of it together with reviews [4]. To encourage independence of the testers, essential for the desired security, a P2P recommendation mechanism is employed. It suggests testers for end-users using various metrics (such as: reviews, connection distance, frequency of usage) [5].

III. INCENTIVES FOR SUPERNODES

In our approach, each human owning a peer can control the traffic supported by her system. Peers found behind NATs require significant support from computers with Internet IP addresses, called supernodes. Users with access to servers that have the needed resources are encouraged to offer supernode services with incentives such as: (a) helping the endpoints of the communication, e.g., based on friendship, (b) helping a cause, e.g., based on the content/topic of the communication, (c) reputation, or (d) the utility brought to the supernode user by the handled data (which can be seen as subsuming the case b) [6]. Figure 2 illustrates the current interface where users can configure the negotiation of their agents with the supernodes.

IV. AD HOC WIRELESS

The wireless ad hoc communication module allows peers to create a unique wireless SSID, DirectDemocracy, at frequency 2.462 GHz, resulting in the cell 46:32:D1:F2:88:67. This cell is common for all peers using the application. The module interface detects wireless interfaces, and can be configured to use any subset of them.

Each of the broadcast queues are preloaded with messages. Configurable policies are used in order to load items of certain types (personal, similar to personal, recent, random, round-robin, requested). The agent will only broadcast items related to specified interests. These queues have specific mechanisms for loading and reloading.

The broadcast client picks items from the various existing queues based on a probability distribution that can be specified by the user. A utility-based scheme is used to automatically optimize this probability distribution in order to maximize the satisfaction of interests of the sender. The utility-based scheme uses information such as: the number of other vehicles traveling in the same direction or in opposite direction, their relative speed, their declared interests, and the worth associated by the user to the dissemination of different types of information [7]. A GPS device can be used to extract the needed information.

V. DEMO ITEMS

The demo focuses on: communication modules (NATs, ad hoc wireless networks, supernodes), GUI control, and plugins. It shows how to manage one’s peer identity and to connect to other peers, how to create or join organizations, how to review other peers, to submit and sign motions and news, and to specify interests and filters. It illustrates how to develop and install plugins, demonstrates available ones, and how to register testers and mirrors for automatic updates.

REFERENCES

Appendix B

Online Survey to Study the
effectiveness of an Incentive-based P2P
Chat System
Skype is free but for its technical and economical survivability it may interfere with your privacy and freedom. In this study, we want to see if it is possible to have a technically survivable system that guarantee user privacy and freedom, and provide similar performance as Skype.

1. Have you ever used Skype?
   - Yes
   - No, I don't use Skype because: (Choose one or more options)
     - I've never heard of Skype software
     - All my friends use something else
     - It is difficult to install
     - It has privacy issues
     - Other (please specify): 

*Note: If you never used Skype, you don't need to answer the rest of the survey just click on the submit button in the end of the survey*

2. How often do you use Skype?
   - Daily
   - Three times a week
   - Once a week
   - Once a month
   - Less than once a month

3. Do you keep Skype running while you are using your computer?
   - Yes, I keep Skype running so I can communicate with my friends while working on my computer
   - No, I quit Skype once I have done communicating with someone

4. For what purposes do you use Skype most? (Choose one or more options)
   - Video calls
   - Skype to Skype voice calls
   - Skype to phone voice calls
   - Online messaging (Instant Messaging)
   - Video conference
   - SMS
   - Other (please specify):

5. How many devices (desktop/laptop/tablet/cell phone) do you have that have Skype installed on them?

The number of devices that I have with Skype installed on them: 2

5.1. For each device (desktop/laptop/tablet/ cell phone) that has Skype installed on it,
answer the following questions:
Note: each column in the table is representing one of the devices you have.

<table>
<thead>
<tr>
<th>Type</th>
<th>Device1</th>
<th>Device2</th>
</tr>
</thead>
<tbody>
<tr>
<td>How long is it connected to the Internet at home?</td>
<td><img src="select_one" alt="Select options" /></td>
<td><img src="select_one" alt="Select options" /></td>
</tr>
<tr>
<td>○ 1 - 3 hours a day</td>
<td>○ 1 - 3 hours a day</td>
<td></td>
</tr>
<tr>
<td>○ 3 - 5 hours a day</td>
<td>○ 3 - 5 hours a day</td>
<td></td>
</tr>
<tr>
<td>○ 5 - 7 hours a day</td>
<td>○ 5 - 7 hours a day</td>
<td></td>
</tr>
<tr>
<td>○ 7 - 9 hours a day</td>
<td>○ 7 - 9 hours a day</td>
<td></td>
</tr>
<tr>
<td>○ More than 9 hours a day</td>
<td>○ More than 9 hours a day</td>
<td></td>
</tr>
<tr>
<td>○ Doesn't use it at home</td>
<td>○ Doesn't use it at home</td>
<td></td>
</tr>
<tr>
<td>What time during the day do you usually leave your device running at home? (Choose one or more options)</td>
<td><img src="select_multiple" alt="Select options" /></td>
<td><img src="select_multiple" alt="Select options" /></td>
</tr>
<tr>
<td>○ Day time [6am - 12pm]</td>
<td>○ Day time [6am - 12pm]</td>
<td></td>
</tr>
<tr>
<td>○ Afternoon [12pm-6pm]</td>
<td>○ Afternoon [12pm-6pm]</td>
<td></td>
</tr>
<tr>
<td>○ Evening [6pm-12am]</td>
<td>○ Evening [6pm-12am]</td>
<td></td>
</tr>
<tr>
<td>○ Late [12am - 6am]</td>
<td>○ Late [12am - 6am]</td>
<td></td>
</tr>
<tr>
<td>○ Doesn't use it at home</td>
<td>○ Doesn't use it at home</td>
<td></td>
</tr>
<tr>
<td>How long is it connected to the Internet at work/school?</td>
<td><img src="select_one" alt="Select options" /></td>
<td><img src="select_one" alt="Select options" /></td>
</tr>
<tr>
<td>○ 1 - 3 hours a day</td>
<td>○ 1 - 3 hours a day</td>
<td></td>
</tr>
<tr>
<td>○ 3 - 5 hours a day</td>
<td>○ 3 - 5 hours a day</td>
<td></td>
</tr>
<tr>
<td>○ 5 - 7 hours a day</td>
<td>○ 5 - 7 hours a day</td>
<td></td>
</tr>
<tr>
<td>○ 7 - 9 hours a day</td>
<td>○ 7 - 9 hours a day</td>
<td></td>
</tr>
<tr>
<td>○ More than 9 hours a day</td>
<td>○ More than 9 hours a day</td>
<td></td>
</tr>
<tr>
<td>○ Doesn't use it there</td>
<td>○ Doesn't use it there</td>
<td></td>
</tr>
<tr>
<td>What time during the day do you usually leave your device running at work/school? (Choose one or more options)</td>
<td><img src="select_multiple" alt="Select options" /></td>
<td><img src="select_multiple" alt="Select options" /></td>
</tr>
<tr>
<td>○ Day time [6am - 12pm]</td>
<td>○ Day time [6am - 12pm]</td>
<td></td>
</tr>
<tr>
<td>○ Afternoon [12pm-6pm]</td>
<td>○ Afternoon [12pm-6pm]</td>
<td></td>
</tr>
<tr>
<td>○ Evening [6pm-12am]</td>
<td>○ Evening [6pm-12am]</td>
<td></td>
</tr>
<tr>
<td>○ Late [12am - 6am]</td>
<td>○ Late [12am - 6am]</td>
<td></td>
</tr>
<tr>
<td>○ Doesn't use it there</td>
<td>○ Doesn't use it there</td>
<td></td>
</tr>
</tbody>
</table>

**6. How many contacts do you have on Skype (the size of the contact list on Skype)?**
- ○ 1 - 20 contacts
- ○ 21 - 50 contacts
- ○ 51 - 100 contacts
- ○ 101 - 200 contacts
- ○ More than 200 contacts

**7. How many different people do you regularly contact by Skype in a month?**
- ○ 1 - 10 contacts
- ○ 11 - 20 contacts
- ○ 21 - 30 contacts
- ○ 31 - 40 contacts
- ○ More than 40 contacts
8. Do you know that Skype is using your computer's resources to handle communication between other Skype software users which has absolutely nothing to do with you. In other words, you give permission to Skype to use your computer for their benefit with no obligation. Carefully take a Look at the following permission you give to Skype when installing first time.

9. If you have the option of stopping Skype from using your computer's resources while still having its benefits; What would you do?

10. Do you know that Skype may use your computer to forward communication between other users on topics that you may not support. (e.g. indecent videos or pictures).

The following 4 statements are designed to assess your behavior when you have more freedom of choice in Skype. For each statement, indicate the degree of your agreement or disagreement based on a scale from 1 (strongly disagree) to 5 (strongly agree):

11. "If Skype allows me to choose the topics and the causes that I want to support, then I will allow Skype to use my computer in order to support only my favorite topics."

12. "If a friend asks for my help to use my computer's resources for accessing Skype (e.g. while travelling), then I will allow Skype to use my computer in order to help my friend."
13. "Skype can utilize my computer's resources if I can get paid"

<table>
<thead>
<tr>
<th>Strongly Disagree (1)</th>
<th>Disagree (2)</th>
<th>Neutral (3)</th>
<th>Agree (4)</th>
<th>Strongly Agree (5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>

14. “If Skype allows me to see and examine the information that passes through my computer, then I will allow Skype to use my computer’s resources”

<table>
<thead>
<tr>
<th>Strongly Disagree (1)</th>
<th>Disagree (2)</th>
<th>Neutral (3)</th>
<th>Agree (4)</th>
<th>Strongly Agree (5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>

15. If you cannot connect one of your Skype friends without help from a 3rd party, what incentives would you agree to pay/offer to these 3rd parties? (Choose one or more options)

- [ ] Paying a certain amount of money
- [ ] Accepting advertisements
- [ ] Allowing others to examine the exchanged messages with your contacts
- [ ] Revealing the topic of the discussion
- [ ] Getting help from friends if possible

Other incentives you may offer (please specify):

Submit Query  Reset
Appendix C

List of Publications
The following are the list of publications that have been produced during the course of this Ph.D. research.

Khalid Alhamed, and Marius Silaghi ”User Freedom: To Be or Not To Be a Supernode” Peer-to-Peer Computing (P2P), 2014 IEEE Fourteenth International Conference on. IEEE, 2014.

Khalid Alhamed, Marius Silaghi, Ihsan Hussien, and Yi Yang, ”Distributed Recommendation of Testers for Software Updates in Agent Systems”, Intelligent Agent Technology, IAT, Atlanta 2013


Marius Silaghi, Song Qin, Khalid Alhamed, Toshihiro Matsui, Makoto Yokoo, Katsutoshi Hirayama, ”P2P Petition Drives and Deliberation of Shareholders”, Open Peer-Reviewed Workshop on Decentralized Coordination (DC 2013), Melbourne, Florida, US, April, 2013
Khalid Alhamed, Marius Silaghi, Ihsan Hussien, Yi Yang, "Security by Decentralized Certification of Automatic-Updates for Open Source Software controlled by Volunteers", Open Peer-Reviewed Workshop on Decentralized Coordination (DC 2013), Melbourne, Florida, US, April, 2013