# "Stacking the Deck" Attack on Software Updates: Solution by Distributed Recommendation of Testers

Khalid Alhamed, Marius C. Silaghi, Ihsan Hussien, Ryan Stansifer, Yi Yang
Florida Institute of Technology

***Abstract*— The discussed "Stacking the Deck" attack and our solution are relevant only to software controlled by loosely constituted communities. 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. We propose a mechanism to reduce the level of trust that users are required to have in the maintainers of free and open-source agent software. In fact, with the proposed method, it is sufficient for the user to trust that his constellation of independent testers are safe from attack, even as all testers may be subject to different attacks.**

**Our solution inserts independent intermediaries (testers) between the developers and the end-users. To encourage independence of the testers, essential for the desired security, a distributed 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 her neighborhood. Metrics of success and experiments for identifying promising parameters are reported.**

## I. Motivation

A free GPL open-source agent system for supporting petitions drives, like DirectDemocracyP2P [1] (DDP2P [http://directdemocracyp2p.net/]), is under heavy development even after its release, and so it needs an automated update mechanism.

Given the main application of such a system, namely petition drives, eventually somebody will use the system to get signatures for a petition that is not liked by, say, BIG MONOPOLY INC. Currently, an easy way 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 have them replaced with people that they control. 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 subtle that it may take a long time for any user to even realize what is happening. For example, an automatic update can 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 lost for ever, and volunteers lack a mechanism warn users. Let's refer this as the *Stacking the Deck Attack*.

Our solution here is to insert independent intermediaries between the developers and the end-users, intermediaries that can eventually detect and then warn or hamper the automatic damage of the whole process. Once a Stacking the Deck Attack is detected, other volunteers can start new branches (given that the code is free open source). This opportunity provided by free open source is why we make the assumption to handle such projects in the rest of the paper.

These intermediaries can detect certain attacks based on independent testing of the software. As such, the assumption of independence is essential for increasing the trust and resilience of the whole process. A way to encourage independence of the intermediaries (in a slow mixing social network expected for DDP2P) is provided by a distributed recommendation mechanism, where testers used by closer but fewer neighbors are given higher ranking. Here we report on this recommendation system.

## II. Background

Decentralized protocols are used for recommending resources such as links and files among agents [2], [3]. Significant work exists on updates of software. General update problems, like breaking dependencies between packages due to a removal of some important feature, buggy updates or incompatible updates, are addressed in [4]. A classification of several known security issues is described in [5], [6]. Sometimes the software handles its own updates, while in other cases the operating system performs the automatic updates via package managers [7], [8]. The issues that have to be addressed by systems for automatic updates to free software can be substantially different from issues with updates to commercial software, where licenses have to be verified [9], [6] but responsibility for the updates quality is centralized by the merchant. A schema for disseminating large updates based on a chain of fragments authenticated efficiently by including the hash of the next fragment in the previous authenticated fragment is proposed in [5]. A technique for asynchronously rekeying secure communication for updates is proposed in [6].

Recent research has already identified the fact that using more then one key can help to improve security of updates systems against attacks. The observation was that when updates are signed with several keys, the work of the attacker

978-1-4799-2902-3/13 $31.00 © 2013 IEEE
DOI 10.1109/WI-IAT.2013.123

IEEE computer society

is more difficult than for breaking a single key. The solution proposed in [10] 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 [10]. The implementation suggested in [10] for the aforementioned idea is to use separate keys for various roles, such as: the content of updates (targets role and delegated targets role), the availability of updates (timestamp role).

## III. Concepts

**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 [11], [12]. Once defined, future releases cannot redefine a previous definition (but rather can add new ones), thereby helping testers to warn users of Stacking the Deck Attacks. 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 the following 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).

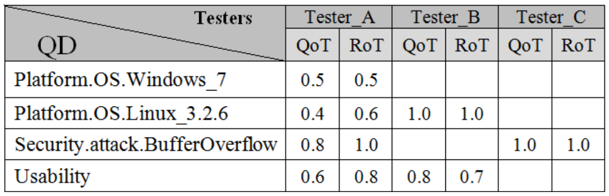
It is essential for the binary builder to be deterministic in order to guarantee that a digital signature generated by a tester for the result $\beta$ of her own compilation of sources $\vec{\Sigma}$ that she analyzed is applicable to the binaries built and distributed by others. If the binary builder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

**Testers.** Our framework brings independent testers into the mechanism for ensuring the quality of free and open-source software (FOSS). The assumptions are that:

- testers can study each release independently.
- testers provide Quality of Test (QoT) and Results of Test (RoT) information
- testers append their signature to the release, signing the data identifying it (version, file names, hashes) and their quality evaluation (QoT, RoT).

*Example 2:* Consider the first QD of Example 1, *"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 I, 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 I, tester_A has tested the binaries and found some flaws, so she specified 0.5 as value for RoT on *support of Windows 7*



| Testers / QD | Tester_A | | Tester_B | | Tester_C | |
|---|---|---|---|---|---|---|
| | QoT | RoT | QoT | RoT | QoT | RoT |
| Platform.OS.Windows_7 | 0.5 | 0.5 | | | | |
| Platform.OS.Linux_3.2.6 | 0.4 | 0.6 | 1.0 | 1.0 | | |
| Security.attack.BufferOverflow | 0.8 | 1.0 | | | 1.0 | 1.0 |
| Usability | 0.6 | 0.8 | 0.8 | 0.7 | | |

TABLE I
QD and Testers certificates on QoT & RoT

**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.

## IV. Recommender System for Testers

Testers play an essential role in our mechanism for auto-updating FOSS build on agent architectures, and 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). The question here is how can agents know about the available testers, and how

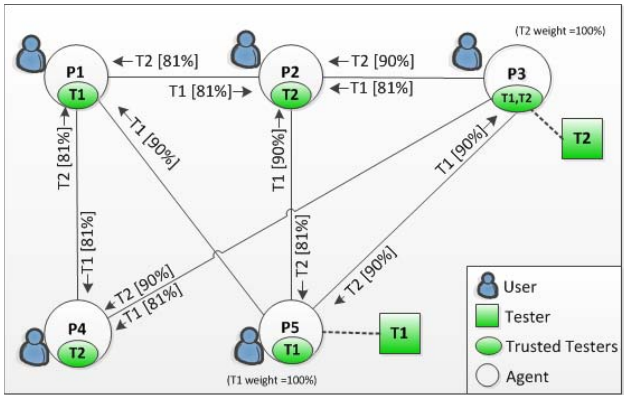


Fig. 1. Overall Architecture of the Recommender System

can agents trust these testers, without a centralized recommendation system that can be taken over by the attacker. We provide a distributed recommendation mechanism, where testers are advertised to other agents. Weights are used to rank recommended testers.

### A. Heuristics for Systems Recommending Testers

While there are various ways to build systems that recommend testers to end-users, we now highlight principles that can help maximize security. The main one refers to the independence of the testers,

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

Without this principle an attacker controlling the recommender system can filter only testers that she controls. Even with a decentralized recommender, the criteria of a recommender can be exploited to focus on a few testers (which being few can be easier attacked). A heuristic to help distribute the trust away from a small kernel, is to take into account proximity (assuming the end-users are themselves distributed reasonably well).

**Heuristic** *[Proximity] As a heuristic for independence of testers one can give priority to testers that are close (or far) to 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.*

### B. Recommender System for testers

For agent systems, such as DDP2P, there exists an intrinsic social network as defined by the connections of each agent (or constituent). In one such scheme, testers used by an agent are recommended to neighboring agents. Each tester being associated with a weight (trust coefficient), 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 that are close to the user in terms of the social network.

By default, the recommendation made to an agent for a tester has the weight given by the maximum value among the weights coming on all its links. Users can overwrite this default for themselves by increasing or decreasing the weight manually. The recommendation is forwarded only if the user manually accepts to use the recommended tester. If a user decides to not 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 to others. This mechanism improves scalability since testers are not propagated infinitely. Users can define and act themselves as testers, or introduce manually testers they personally know and trust. Based on this scheme, their neighbors receive high recommendations for them.

For example, assuming the amortization coefficient is 90% (i.e., the trust coefficient is reduced with 10% for each new link in the chain of recommendation), the obtained recommendation in an agent system is shown in Figure 1. As shown, there are six agents (P1,...,P6) 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, whom she also assigns a weight of 100%. Both P3 and P5 pass their selected testers information to neighboring agents. In Figure 1, P3 announces T2 to her neighbors agents P2, P4 and P5 which see T2 recommended with the weight 90% (0.9*100%). Also, P5 recommended T1 to her neighbors agents P1, P2 and P3 which see the weight 90%. Based on these recommendations, P2 and P4 have decided to use T2 as trusted tester and forward T2's information to P1 which see the associated weight 81% (0.9 * 90%). In addition, P1 has decided to use T1 as a trusted tester (P1 had the choice to use T2[81%] or T1[90%] or both). However, P3 has decided to use T1 as trusted tester beside T2.

### C. Messages Exchanged

To exchange the information about testers between agents, we are now formalizing the concept of the tester item. We assume that global identifiers (GIDs) are used to uniquely specify agents and testers (as done by common agent platforms).

A tester item is a tuple $\langle \tau, \mathcal{A}, \mathcal{W}, d, \mathcal{P}, \mathcal{S} \rangle$, where $\tau$ is the GID (public key) of the tester, $\mathcal{A}$ is the address where the tester can be contacted for retrieving her mirrors, $\mathcal{W}$ is the weight of the recommendation made to the sender by the system (amortized aggregated value it received from her neighbors, and which can have been manually overwritten by the sender), $d$ is the timestamp of the given weight, $\mathcal{P}$ is the GID of the sending agent and $\mathcal{S}$ is the digital signature with which the sender authenticates the provided weight: $\mathcal{S} = SIGN(SK(\mathcal{P}), \langle \tau, \mathcal{A}, \mathcal{W}, d \rangle)$.

Only the newest tester item is stored for a given pair $\langle\tau, \mathcal{P}\rangle$, as per the timestamp $d$. Each received tester item from each agent is stored along with the arrival date. This arrival time is used during synchronization with neighbors, to keep track of already exchanged tester items.

Testers manually introduced by the user are given a fixed weight (e.g. 100%). Given a set of $n$ tester items $\langle\tau_i, \mathcal{A}_i, \mathcal{W}_i, d_i, \mathcal{P}_i, \mathcal{S}_i\rangle$ received from agents, the current weight of the recommendation made to the user for tester $\tau_i$ is computed as:

$$f \cdot (1 - \frac{\#\{i|\tau_i = \tau_j\}}{K \cdot n}) \cdot max\{\mathcal{W}_j|\tau_i = \tau_j\}. \tag{1}$$

where $f$ is the amortization factor, $K$ is a factor modeling the trade-off between proximity and diversity ($K \geq 1$, typically $K = 2$), $\{a|b\}$ denotes the set of elements $a$ for which the condition $b$ holds, $\#A$ is the cardinality of the set $A$, and $\max(A)$ is the maximum numerical element of the set $A$.

In case a user decides to revert to a previous software version and invocates dissatisfaction with some properties in the QoD of the system, a penalty $p$ is added to the weight of the testers that have recommended low scores on those properties and is subtracted from the weight of the testers that have recommended high scores for those properties. The penalty is weighted proportional with the deviation of their QoT*RoT from 0.5: $\mathcal{W}_{final} - \mathcal{W}_{initial} \sim 0.5 - QoT * RoT$. initial For example, if all neighbors recommend only one and same tester, Equation 1 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, $\frac{Kn-1}{Kn}$, of the weight received from that neighbor, amortized with $f$.

**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 does still 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.

While a user does not manually select testers from the recommended list, and does not manually introduce any, the system automatically uses the top recommended fraction of the recommended testers (up an upper limit, in our experiments we set this fraction to 10 testers). Once a user manually selects testers based on the available recommendations, then her list of tester is no longer automatically classified. Each time a new tester is selected, the tester is automatically contacted at the provided address for a list of mirrors using her reviews.

### D. The Frequency of Testers Replacement

All the testers known by an agent are stored in a list $KT$, sorted by their weight. Each agent maintain a list of used testers, $UT$. Advanced users populate this list manually while remaining users get their $UT$ list populated automatically by the distributed recommender system. Based on the computation model in Equation 1, agents evaluate the weight of the testers in $KT$ based on the the recommendations received from their links. With a certain frequency (e.g., daily) an agent recomputes the weights of each of the testers it knows and sends a message to its links, describing the testers in its $UT$, as well as a number $k^t$ of other top testers in $KT$. The more different tester configurations are adopted by an agent, the higher is the risk that a configuration controllable by an attacker is eventually selected. In order to reduce the amount of experienced tester configurations an agent only switches a single tester at a time, and only with a certain probability, $p$.

## V. Analysis

### A. Requirements on Testers

Testers can release signed reviews for an analysis of the software based on its sources. The tester can also issue reviews for binaries compiled by others, but then she cannot be sure about her analysis of the source code (as she cannot verify that the binary she studies is indeed based on the source that she can access).

If the tester want to release reviews for a given binary release (coupled with results of direct inspection of the sources), the only requirement is that they use a *binary builder* (a compilation process that leads to the same binaries). That is typically achieved if they compile with the same compiler options, and with the same version of the compiler. For example, with Java, the same binary is achieved from any machine and distribution of Linux if the same java compiler is used. In general, testers contributing to a given binary release *do not need to have identical testing machines*.

The advantage of a Java binary is that it runs on any operating system, and therefore signed test results on any platform (e.g., Linux) are automatically applicable to binaries running an many platforms (Windows, Mac).

### B. Security Evaluation

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

## VI. Architecture

A binary release of open source software undergoes four processes (see Figure 2):

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* [13]. 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

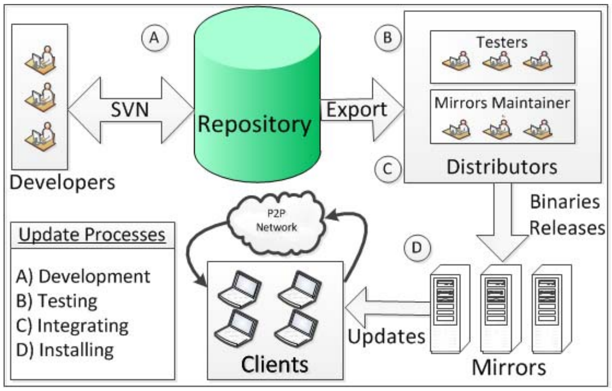


Fig. 2. Overall Architecture of Integrated Development, Testing and Updates

| Symbol | Description |
|---|---|
| $\vec{\Sigma}$ | release sources |
| $\nu$ | version identifier (i.e., 1.2.0) |
| $\vec{\Phi}$ | quality definitions added by a developer |
| $d$ | release date |
| $\tau$ | tester ID |
| $\beta$ | binary software |
| $\vec{\eta}$ | information for release files |
| $\varepsilon$ | release building parameters |
| $\epsilon$ | Boolean flag, false for $\varepsilon = \perp$ |
| $t$ | the date of the test data |
| $\vec{\Upsilon}$ | quality definitions added by a tester |
| $\vec{\Theta}$ | vector of Qualities of Tests |
| $\vec{\Psi}$ | vector of the Result of Tests |
| $\mathcal{A}$ | tester address |
| $\mathcal{W}$ | tester weight |
| $\vec{\Delta}$ | testers trusted by user |
| $\mathcal{P}$ | message sender |
| $\mathcal{S}$ | digital signature |
| $\delta$ | secret key |
| $\perp$ | empty value (i.e. null) |

TABLE II
TABLE OF SYMBOLS

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 will 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 I). Similarly, a tester specialized on *Linux* can test properties related to Linux (see tester_B in Table I). 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 will guarantee 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 $\langle \tau, \beta, \vec{\eta}, \nu, \varepsilon, \vec{\Phi}, \vec{\Upsilon}, \vec{\Theta}, \vec{\Psi}, d, \mathcal{S} \rangle$ 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 $\mathcal{S} = SIGN(\delta, \langle \vec{\eta}, \nu, \varepsilon, \vec{\Phi}, \vec{\Upsilon}, \vec{\Theta}, \vec{\Psi}, d \rangle)$ 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 $\varepsilon$, it which case it is issued with a special value for $\varepsilon$, $\varepsilon = \perp$ (empty). The signature is this case is computed for $\vec{\eta} = \perp$. $\mathcal{S} = SIGN(\delta, \langle \perp, \nu, \perp, \vec{\Phi}, \vec{\Upsilon}, \vec{\Theta}, \vec{\Psi}, d \rangle)$

C) *Integration process.* A mirror maintainer integrates $\beta$ as obtained from a tester with the quality reviews, each of them of type $(\tau_i, \beta, \nu, \varepsilon, \vec{\Phi}, \vec{\Upsilon}_i, \vec{\Theta}_i, \vec{\Psi}_i, d, \mathcal{S}_i)$, 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 over the available testers. If a tester issue reviews both for $(\nu, \varepsilon)$ and for $(\nu, \perp)$, keep only the one for $(\nu, \varepsilon)$. 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 $\{\langle \tau_i, \vec{\Upsilon}_i, \vec{\Theta}_i, \vec{\Psi}_i, \epsilon, t, \mathcal{S}_i \rangle\}$, $\tau_i$ 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. Trusted testers for novice users are selected using the P2P recommender system embedded into the application, while advanced users specify their preferred testers. If automatic updates are enabled and user-defined criteria concerning required tester support and minimal quality levels are satisfied, then the binary will be 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.

## VII. 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 evaluteUpdate() 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: $\langle\vec{\eta},\nu,\varepsilon,\vec{\Phi},d,\Gamma\rangle$
- The user predefined conditions for each quality definition, aggregated in the tuple: $\langle w,c,\mu,\vec{\Delta}\rangle$ 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), $\vec{\Delta}$ 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 reject the update. The total weight of the trusted testers supporting this update and their count is computed and stored in the variables $total\_wt$ and $cnt\_testers$, respectively (Lines 4, 5, 18 and 22). The combined quality of tests and results are maintained in the vectors $crt\_QoT$ and $crt\_RoT$ (see Lines 6, 7, 19 and 21). A sample combination function for $QoT$ is $max$ and for $RoT$ is $min$. $crtWeight$ returns the weight of tester given 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, $\vec{\Delta}$, 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 PK($\tau$). If the signature of the review is not valid then that review is excluded from $\Gamma$ (Line 24). Function getRequiredTesters() return a list of the testers without whose supporting reviews the user refuses any automatic update (Line 25).

**Algorithm 1:** End-user algorithm for accepting automatic updates

```
1  function $evaluateUpdates(\langle\vec{\eta},\nu,\vec{\varepsilon},\Phi,d,\Gamma\rangle, \langle w,c,\mu,\vec{\Delta}\rangle) \longrightarrow Boolean$
2      if ($\nu$ not newer than currentVersion) then
3          return false;
4      $total\_wt \leftarrow 0$;
5      $cnt\_testers \leftarrow 0$;
6      $crt\_QoT \leftarrow [0,...0]$;
7      $crt\_RoT \leftarrow [0,...0]$;
8      remove double occurrence of testers in $\Gamma$ (prefer occurrences with newer date $t$ and more specific, $\varepsilon \neq \bot$);
9      foreach $(\langle\tau,\vec{\Upsilon},\vec{\Theta},\vec{\Psi},\epsilon,t,\mathcal{S}\rangle \in \Gamma)$ do
10         $\varepsilon' \leftarrow \varepsilon$; $\vec{\eta}' \leftarrow \vec{\eta}$;
11         if $(\neg\ \epsilon)$ then $\varepsilon' \leftarrow \bot$; $\vec{\eta}' \leftarrow \bot$;
12         if $(\tau \notin \vec{\Delta})$ then
13             $\Gamma \leftarrow \Gamma \setminus \{\langle\tau,\vec{\Upsilon},\vec{\Theta},\vec{\Psi},\mathcal{S}\rangle\}$;
14             continue;
15         if $(revoked(PK(\tau)))$ then continue;
16         $r \leftarrow verify(PK(\tau), \langle\vec{\eta}',\nu,\varepsilon',\vec{\Phi},\vec{\Upsilon},\vec{\Theta},\vec{\Psi},d,\mathcal{S}\rangle)$;
17         if $r = true$ then
18             $total\_wt \leftarrow total\_wt + getWeight(\tau,\vec{\Theta})$;
19             $crt\_QoT \leftarrow combineQoT(crt\_QoT,\vec{\Theta},\tau)$;
20             $crt\_RoT \leftarrow$
21                     $combineRoT(crt\_RoT,\vec{\Psi},\vec{\Theta},\tau)$;
22             $cnt\_testers \leftarrow cnt\_testers + 1$;
23         else
24             $\Gamma \leftarrow \Gamma \setminus \{\langle\tau,\vec{\Upsilon},\vec{\Theta},\vec{\Psi},\mathcal{S}\rangle\}$;
25     if $(getRequiredTesters() \not\subseteq \Gamma)$ then return false;
26     if $(crt\_QoT \ngeq getRequiredQoT())$ then
27         return false;
28     if $(crt\_RoT \ngeq getRequiredRoT())$ then
29         return false;
30     if $(\mu = WEIGHT)$ then
31         return $(total\_weight \geq w)$;
32     if $(\mu = COUNT)$ then
33         return $(cnt\_testers \geq c)$;
```

Function getRequiredQoT() (used in Line 26) 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 26 where each entry of crt_QoT must be greater or equal to the corresponding required value. Function getRequiredRoT() (used in Line 28) 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

$crt\_RoT$ is smaller than the corresponding entry in the result of $getRequiredRoT$, 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 30) or based on their total number (Line 32).

## VIII. Experiments

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

- 10000 agents ($P_0..P_{9999}$) divided in 100 neighborhoods (of 100 agents each). For example $P_0..P_{99}$ are in neighborhood $N_1$, $P_{100}..P_{199}$ in neighborhood $N_2$, etc.
- Each agent is linked to 50 other agents. 48 of these links are within the neighborhood of the agent (only 2 of the links are out of the neighborhood).
- 100 testers ($T_0..T_{99}$), each introduced by an agent as follows: $P_{100i}$ manually selects $T_i$ for usage with weight 100%.
- Each agent not manually selecting its testers will automatically use at most 10 testers, 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 agent: e.g., each day). In each round of the simulation, all agents synchronously evaluate recommendations from their links and decide new usage. We run 100 such rounds for each experiment and then analyze the results.

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

- **Distribution**. The distribution (usage) of testers among agents should be correlated to their perceived quality (as per an aggregated value of the weights provided by the agents manually introducing them). However, the differences should not be large, to avoid giving an overwhelming power to attackers that succeed to manipulate their weight.
- **Local Stability**. The set of testers used by a given agent should be stable (not changing frequently). If this set changes frequently then this increases the likelihood that an attacker eventually get the opportunity (favorable configuration of testers) to manipulate a given agent into updating to a doctored version. The local stability evaluates the frequency of switching testers for individual agents.
- **Global Stability**. We want to avoid having high local stability for some agents while other agent switch their testers frequently. We measure the global stability as the average of the local stability metrics for all agents.
- **Casualties**. We want to reduce the number of agents taken over by the attacker. This number is computed by counting how many agents eventually use a configuration of testers that can be controlled as whole by the attacker (e.g., the majority of the testes are controlled by the attacker).

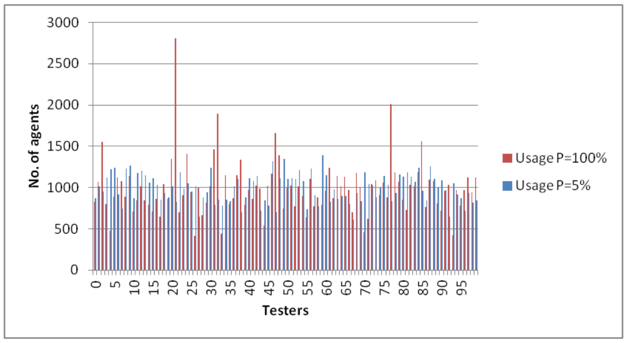


Fig. 3. Number of users trusting each tester after 100 rounds. The users are supposed to follow the recommender system

Once an agent 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 agent. 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 agents to avoid them, since now those agents are too frequently used; on the other side, recommending un-controlled testers wastes opportunities to disseminate controlled ones). Given this dilemma of the attacker, we currently let controlled agents to behave as the other agents. Also, 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.

### A. Evaluation of Distribution

In the first reported experiment we show the distribution (usage) of testers among agents after running the simulation with 100 rounds. In this experiment, each agent switches to the top 10 testers recommended to it in the round. The other parameters are $k = 2$ and $f = 0.9$. As one can see in Figure 3 the standard deviation for tester usage is 331. For example (in the worst scenario), assuming an attacker agent introduced the malicious tester $T_{21}$, one ends up with 2803 agents using $T_{21}$ (maximum usage in this experiment).

We also run an experiment with $p = 5\%$, $k = 2$ and $f = 0.9$. One can see in Figure 3 that the standard deviation for tester usage is reduced to 173. Now (in the worst scenario) when an attacker agent introduce a tester $T_{59}$, then $T_{59}$ is used by 1393 agents (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 agents where only 50 agents are from the same neighborhood $N_{86}$.

### B. Global Stability

In this experiment we compare two values for parameter $f$ based on global stability. Simulations with 50 rounds each are

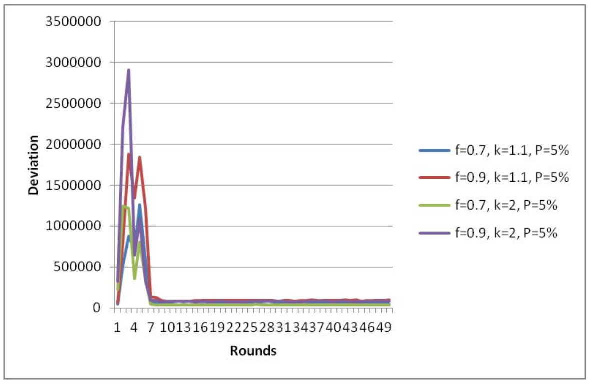


Fig. 4. Global Stability: where $k \in \{1.1, 2\}$ and the probability of replacement is $P = 5\%$ [comparing parameter $f = 0.9$ and $f = 0.7$]

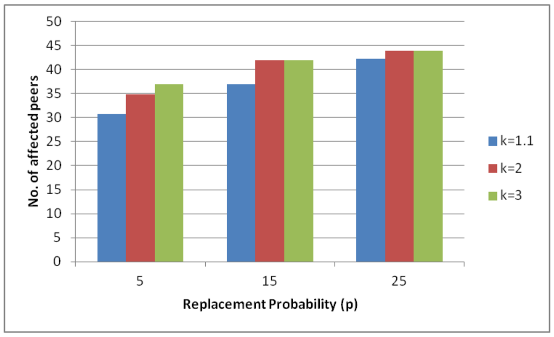


Fig. 5. Casualties after 50 rounds where f=0.7.

performed for the the parameters $f = 0.7$ and $f = 0.9$. Here we use $p = 5\%$. In Figure 4 we report results for $k = 2$ and $k = 1.1$. The experiment suggests that the best parameters values in these situations are $f = 0.7$ and $k = 1.1\%$. More experiments with additional values and more instances are planned for the final version.

### C. Casualties

An experiment based on simulations with 50 rounds each, for the the parameters $f = 0.7$ and $k = 1.1$. is used to detect the best value for the parameter $p$. The results in Figure 5 are averaged over 10 instances for each of the values 5%, 15%, and 25% for the parameter $p$. The experiment suggests that the best parameter in this situation in $p = 5\%$. More experiments with additional values and more instances will be performed in the future.

## IX. Conclusions

We address the problem of free open source agent software in applications of strategic importance (like petition drives). We are concerned that such software can be the target of an attack from well funded attackers, attack that we coin under the name: *"Stacking the Deck"* attack. This attack consists of orchestrating the transfer of the leadership of the development team to people that the attacker controls, 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. 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.

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.

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

## References

[1] M. C. Silaghi, K. Alhamed, O. Dhannoon, S. Qin, R. Vishen, R. Knowles, I. Hussien, Y. Yang, T. Matsui, M. Yokoo, and K. Hirayama, "DirectDemocracyP2P — decentralized deliberative petition drives —," in *Proceedings of IEEE P2P*, Trento, September 2013.

[2] M. Mordacchini, R. Baraglia, P. Dazzi, and L. Ricci, "A p2p recommender system based on gossip overlays (prego)," in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, 2010, pp. 83–90.

[3] Z. Yu and F. Liu, "Trust-based recommender system in p2p network," in *Communication Systems and Network Technologies (CSNT), 2012 International Conference on*, 2012, pp. 423–426.

[4] O. Crameri, R. Bianchini, W. Zwaenepoel, and D. Kosti?, "Staged deployment in mirage, an integrated software upgrade testing and distribution system," in *In Proceedings of the Symposium on Operating Systems Principles, Bretton Woods*, 2007.

[5] P. E. Lanigan, R. Gandhi, and P. Narasimhan, "Sluice: Secure dissemination of code updates in sensor networks," in *26th IEEE International Conference on Distributed Computing Systems*, November 2006.

[6] D. K. Nilsson, T. Roosta, U. Lindqvist, and A. Valdes, "Key management and secure software updates in wireless process control environments," in *WiSec '08: Proceedings of the first ACM conference on Wireless network security*. New York, NY, USA: ACM, 2008, pp. 100–108.

[7] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman, "A look in the mirror: attacks on package managers," in *Proceedings of the 15th ACM conference on Computer and communications security*, ser. CCS '08. New York, NY, USA: ACM, 2008, pp. 565–574. [Online]. Available: http://doi.acm.org/10.1145/1455770.1455841

[8] Greg and Mark, "Update engine: Software updating framework for mac os x," http://code.google.com/p/update-engine/, 2013.

[9] W. Adi, A. Al-Qayedi, K. Negm, A. Mabrouk, and S. Musa, "Secured mobile device software update over ip networks," in *SoutheastCon*, 2004, pp. 271 – 274.

[10] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine, "Survivable key compromise in software update systems," in *CCS'10*, 2010, p. 61.

[11] R. Dromey, "Cornering the chimera [software quality]," *IEEE Software*, vol. 13, no. 1, pp. 33 –43, jan. 1996.

[12] D. M. Nichols and M. B. Twidale, "The usability of open source software," in *First Monday, volume 8, number 1*, 2003.

[13] T. Dinh-Trong and J. Bieman, "Open source software development: a case study of freebsd," in *IEEE Int. Symposium on Software Metrics*, 2004, pp. 96–105.