Security by Decentralized Certification of Automatic-Updates for Open Source Software controlled by Volunteers

Khalid Alhamed and **Marius C. Silaghi** and **Ihsan Hussien** and **Yi Yang**
Florida Tech

Abstract

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 propose a mechanism to reduce the level of trust that users are required to have into repositories for open source software that is maintained by volunteers. In fact, with the proposed method, it is sufficient for the user to trust that his flexibly specified constellation of independent testers are safe to each given attack, even as all may be subject to different attacks. The interesting configuration when any majority t out of n `testers` of the given user's choice have to be believed safe, is just a special case. A new integrated framework of open source development, testing, distribution and updating is defined, implemented and made available.

A `tester` is a person that builds and tests an existing source code revision from a repository, and then distributes a signed binary release of it, tagged with a Quality of Test (QoT) and a Result of Test (RoT). An ontology for the QoT and RoT is defined and managed by the developers of the source code, and is fixed at each given revision. The final management of the mirrors distributing binaries tested by several testers is done by the mirror maintainers, who may or may not be from among the trusted testers. Some testers can be defined as reference (a new release is not automatically accepted without their signature and minimum RoT).

Introduction

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 from a user-defined constellation (e.g., fraction) out of a set of testers that the user trusts. An essential feature of any software application is the provision of an automatic updates 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 troubles on famous applications like Apple TV (Bonnington 2012). 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 updates system and block any opportunity for a future automatic fix. Attacks on the updates system can replace an application with a similarly looking malware, and divert any further updates from the original source.

Traditionally testers announce their results directly to developers and/or publish them in specialty magazines. With automatic updates, this may already be too late. With open source software this problem is even more acute, since development can be transferred from one group of volunteers to another, leading to discontinuities in support of features and in general vision of the project. 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.

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 attack from the perspective of those users. This attack is even more likely when the application is passed from some developers to others.

If an attacker manages to tamper source files in a repository used for open source development, it can achieve a powerful attack against all new users of the system and all users of a classical automatic updates mechanisms. This is what actually happens in the aforementioned case of discontinued features.

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, disappearance of features, as well as praises of 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 then 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. The selection of the testers the user trusts can be based on personal experience or on external reputation systems. Further, users may specify that automatic upgrades should be executed only if a certain constellation (e.g., majority) of her trusted testers provide 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 (Bellissimo, Burgess, & Fu 2006). 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" (Cappos *et al.* 2008), 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. After we describe the state of the art in the next section, we will 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.

Background

Significant work exists both on updates of software and updates of documents (Bellissimo, Burgess, & Fu 2006; Bertino *et al.* 2003). Sometimes the software handles its own updates, while in other cases the operating system performs the automatic updates via package managers (Cappos *et al.* 2008; Greg & Mark 2013). 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 (Adi *et al.* 2004; Nilsson *et al.* 2008) 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 (Lanigan, Gandhi, & Narasimhan 2006). A technique for asynchronously rekeying secure communication for updates is proposed in (Nilsson *et al.* 2008).

A set of attacks against updates systems described in (Cappos *et al.* 2008) include: *arbitrary package* (replace legitimate with attacker package), *replay attacks* (send older package instead of a new one), *freeze attacks* (block information about new updates), *extraneous dependencies* (induce user to download attacking package) and *endless data* (crash client system by sending him a huge file).

A set of general purpose security principles identified in previous research as relevant to updates is composed of: *end-to-end authentication* (Bellissimo, Burgess, & Fu 2006), *responsibility separation* (based on roles), *multi-signature trust* (role-based signatures, each of them potentially using threshold-based signature schemes), both *explicit* (CRL/OCSP-based) and *implicit* (expiration date-based) *revocation* (Myers *et al.* 1999), and *minimization of risks* (off-line storage for secret keys) (Samuel *et al.* 2010).

Recent research has already identified the fact that using more then one keys 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 is more difficult than for breaking a single key. The solution proposed in (Samuel *et al.* 2010) 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 (Samuel *et al.* 2010). The implementation suggested in (Samuel *et al.* 2010) 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).

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 updates approach (Cadar & Hosek 2012). This approach introduces strong requirement on the whole code development (where all different versions have be designed to coexist and share data), and the provided prototype requires the capture of all system calls. The synchronization of the different program versions when data has to be sent or received from the network is a significant challenge. The resulting coupling between the old and new versions may be so strong that it can alter the behavior of both versions. Moreover, if the weaknesses of the new versions are not immediately visible (security issues, etc), then they may not be obvious to the user in useful time (before dropping the old version). Nevertheless, this approach can be used in conjunction with the tester-provided recommendations proposed here.

Threat Model

In this paper we discuss the cases when attackers can perform the following feats:

- 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 (Dolev & Yao 1981).
- The attacker can take control over the computers and secret keys of certain testers that build binaries and certify the quality of releases.

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, commercial software developers can gain more trust from their users and can get help in detecting and fixing errors.

There exists 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).

Under certain licenses (GPL, AGPL, LGPL,..), users are allowed, free of charge, to use, distribute, change and extend FOSS. 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 of source files. Volunteers use mailing lists for discussing and coordinating (Antwerp & Madey 2008). Among the most famous repositories one finds: `github.com`, `code.google.com` and `sourceforge.net`.

Repositories: Some of the main features and functions of a repository are to:

- `export` (used by programmers and distributors to extract releases).
- `checkin` (used by programmers to submit contributions).
- `releases` (used by programmers to tag snapshots).

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 (Dromey 1996; Nichols & Twidale 2003).

Example 1 *The DDP2P software has as claimed qualities:*

- *support of Windows 7.*
- *support of Linux version 3.2.6.*
- *resistance to buffer overflow attacks.*
- *easy to learn and use (usability).*

Binary Builder: A binary builder is a deterministic function:

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

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

Testers: Our framework brings independent testers in the center of the mechanism for ensuring quality of 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 *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, 0.5: only binaries were tested, 1: binaries were tested and source was inspected). In Table 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 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

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 1: QD and Testers certificates on QoT & RoT

Quality Review: The quality review[1] (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.

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 *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*.

Architecture

A binary release of open source software undergoes four processes (see Figure 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* (Dinh-Trong & Bieman 2004). To help testers and users tune their expectations for the new release, developers provide a set

[1]We thank Cem Kaner for the perspective of the testing community on the difference between quality review and quality certification.

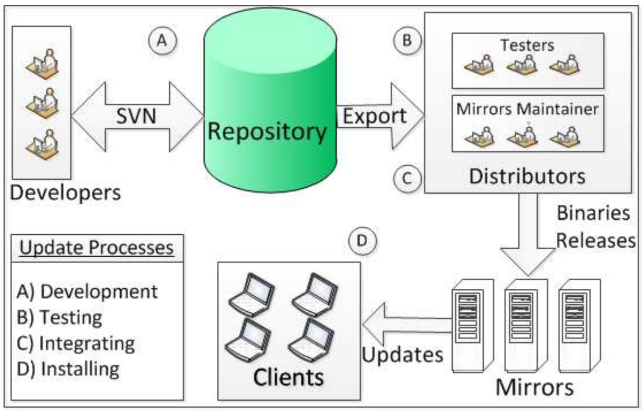


Figure 1: Overall Architecture of Integrated Development, Testing and Updates

Symbol	Description
$\vec{\Sigma}$	release sources
ν	version identifier (i.e., 1.2.0)
$\vec{\Phi}$	quality definitions (see concepts section)
d	release date
τ	tester ID
β	binary software
$\vec{\eta}$	information for release files
ε	release building parameters
ϵ	Boolean flag, false for $\varepsilon = \perp$
t	the date of the test data
$\vec{\Upsilon}$	quality definitions added by tester
$\vec{\Theta}$	vector of Qualities of Tests
$\vec{\Psi}$	vector of the Result of Tests
$\mathcal{S}$	digital signature
δ	secret key
$\perp$	empty value (i.e. null)

Table 2: Symbols Description Table

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, ν is the version identifier, $\vec{\Phi}$ represents the quality definitions and d is the release date.

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 1). Similarly, a tester specialized on *Linux* can test properties related to Linux (see tester_B in Table 1). Each tester compiles and builds her own binaries from the source, to be able to 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 τ is the ID of this tester, β is the binary software, $\vec{\eta}$ is a set of files information including (file names, size and hash values of files content), ε 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 δ 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 ε, it which case it is issued with a special value for ε, $\varepsilon = \perp$ (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 β as obtained from a binary builder 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 (ν, ε) 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 (ν, ε) and for $(\nu, \perp)$, keep only the one for (ν, ε). 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 Γ is a set of tuples $\{\langle\tau_i, \vec{\Upsilon}_i, \vec{\Theta}_i, \vec{\Psi}_i, \epsilon, t, \mathcal{S}_i\rangle\}$, τ_i is the ID of tester i, ϵ 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 (ν, ε) 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 Γ_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 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 (see Algorithm 1 in next section). If automatic updates are disabled, users can inspect the quality reviews and make their decision.

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. 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, 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, μ 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 ε is found relevant for the current system, the algorithm compares the current software version ($currentVersion$) with the newly received update version (ν). 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 Γ (Line 9). If a tester's identifier, τ (digest of its public key), is not found in the list of trusted testers, $\vec{\Delta}$, then its review is excluded from Γ (Lines 12 and 13). The revocation status of the public key from τ 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(τ). If the signature of the review is not valid then that review is excluded from Γ (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($\langle \vec{\eta}, \nu, \varepsilon, \vec{\Phi}, d, \Gamma \rangle$, $\langle w, c, \mu, \vec{\Delta} \rangle$) $\longrightarrow Boolean$
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,
       $\varepsilon \neq \perp$);
9      foreach ($\langle \tau, \vec{\Upsilon}, \vec{\Theta}, \vec{\Psi}, \epsilon, t, \mathcal{S} \rangle \in \Gamma$) do
10         $\varepsilon' \leftarrow vaco$; $\vec{\eta}' \leftarrow \vec{\eta}$;
11         if (not $\epsilon$) then $\varepsilon' \leftarrow \perp$; $\vec{\eta}' \leftarrow \perp$;
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 ($crt\_QoT \not\geq getRequiredQoT()$) then
27         return false;
28     if ($crt\_RoT \not\geq getRequiredRoT()$) 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 μ, trusted testers can be evaluated either based on their total weight (Line 30) or based on their total number (Line 32).

Updates Protocol

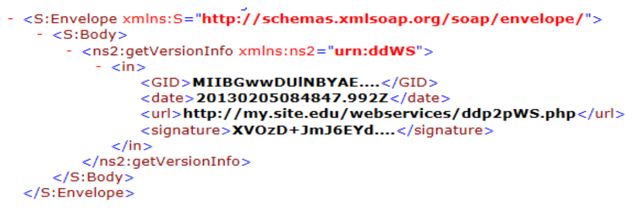
To download updates, a software repeatedly connects to each mirror, url, from its set of preferred mirrors (M). On each connection, the software sends a message **update_request** taking as parameters the tuple $\langle ClientID, url, d, ver, crt, \mathcal{S}\rangle$ (see procedure onClock() in Algorithm 3). In our implementation, procedure OnClock() is set to be called at fix intervals of time. Here $ClientID$ 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 when 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 $\mathcal{S}$ is computed as $\mathcal{S} = SIGN(SK(ClientID), \langle url, d, ver, crt\rangle)$, $SK(ClientID)$ being a notation specifying the secret key of the client identified by $ClientID$.

Also, the server can probabilistically verify the signature in the **update_request** (see Algorithm 2 - Line 2). The mirror server can log requests to enable blacklisting DoS attackers (see Algorithm 2 - Line 6). To mitigate DoS attacks, the server only answers requests where its address is the same as the url parameter in the request (Algorithm 2 - Line 5). If a client sends an invalid signature $\mathcal{S}$ or invalid parameter url, blocking her IP has to be made only when IP spoofing attacks can be ruled out (e.g., with TCP connections). When the difference between the time at the mirror and the d parameter is larger than a reasonable threshold max_skew, then the server replies with a message **time_fault** passing as parameter its actual time (see Algorithm 2 - Line 7). A 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 β. 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 β, based on the Algorithm 1. The actual detailed formats of the exchanged messages are described in Figures 2 and 3.

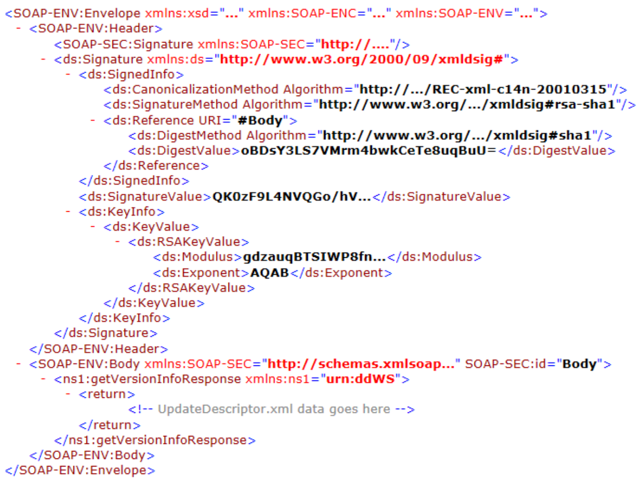
Data Formats for Information Exchange

In this section, we explore the structure of the data exchange between clients and mirrors. First, we identify the elements in the Updates 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).



```
- <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  - <S:Body>
    - <ns2:getVersionInfo xmlns:ns2="urn:ddWS">
      - <in>
          <GID>MIIBGwwDUlNBYAE....</GID>
          <date>20130205084847.992Z</date>
          <url>http://my.site.edu/webservices/ddp2pWS.php</url>
          <signature>XVOzD+JmJ6EYd....</signature>
        </in>
      </ns2:getVersionInfo>
    </S:Body>
  </S:Envelope>
```

Figure 2: Updates Request



```
<SOAP-ENV:Envelope xmlns:xsd="..." xmlns:SOAP-ENC="..." xmlns:SOAP-ENV="...">
 - <SOAP-ENV:Header>
     <SOAP-SEC:Signature xmlns:SOAP-SEC="http://...."/>
   - <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
     - <ds:SignedInfo>
         <ds:CanonicalizationMethod Algorithm="http://.../REC-xml-c14n-20010315"/>
         <ds:SignatureMethod Algorithm="http://www.w3.org/.../xmldsig#rsa-sha1"/>
       - <ds:Reference URI="#Body">
           <ds:DigestMethod Algorithm="http://www.w3.org/.../xmldsig#sha1"/>
           <ds:DigestValue>oBDsY3LS7VMrm4bwkCeTe8uqBuU=</ds:DigestValue>
         </ds:Reference>
       </ds:SignedInfo>
       <ds:SignatureValue>QK0zF9L4NVQGo/hV...</ds:SignatureValue>
     - <ds:KeyInfo>
       - <ds:KeyValue>
         - <ds:RSAKeyValue>
             <ds:Modulus>gdzauqBTSIWP8fn...</ds:Modulus>
             <ds:Exponent>AQAB</ds:Exponent>
           </ds:RSAKeyValue>
         </ds:KeyValue>
       </ds:KeyInfo>
     </ds:Signature>
   </SOAP-ENV:Header>
 - <SOAP-ENV:Body xmlns:SOAP-SEC="http://schemas.xmlsoap..." SOAP-SEC:id="Body">
   - <ns1:getVersionInfoResponse xmlns:ns1="urn:ddWS">
     - <return>
           <!-- UpdateDescriptor.xml data goes here -->
       </return>
     </ns1:getVersionInfoResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 3: Updates Response

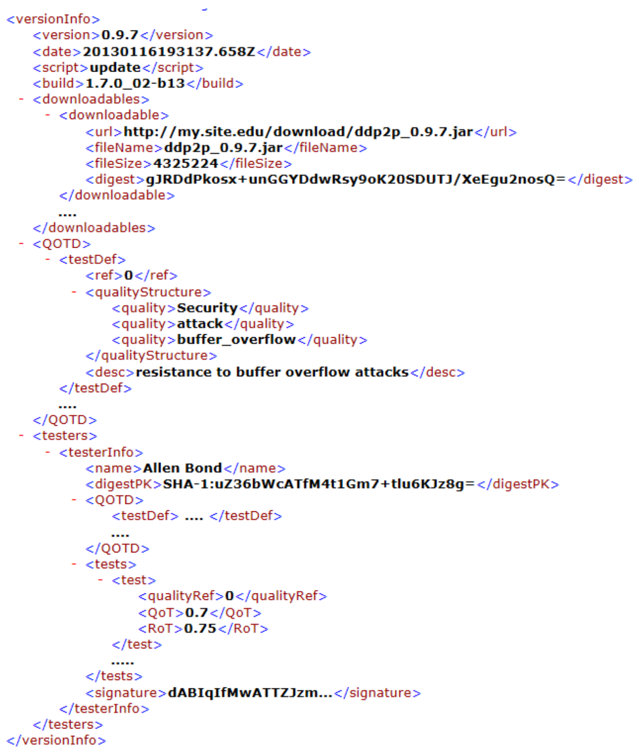
Algorithm 2: Mirror Server

```
1  on update_request $(IP, \langle ClientID, url, d, ver, crt, \mathcal{S}\rangle$ do
2      if (! probabilisticVerification($\mathcal{S}$)) then
3          block IP;
4          return FAIL;
5      if (url $\neq$ myURL) then block IP and return FAIL;
6      log($ClientID, url, d, \mathcal{S}$);
7      if ( $|d - time()| >$ threshold ) then
8          send time_fault(time()) to IP and return FAIL;
9      if ( ver $\neq$ NULL ) then
10         if ( having compatible(ver) ) then
11             send update_response(ver) to IP;
12             return TRUE ;
13         else
14             send version_absent to IP ;
15             return FAIL;
16     if ( latestVersion() newer than crt ) then
17         send update_response(latestVersion()) to IP;
18         return TRUE;
19     send version_absent to IP and return FAIL;
```

The format of the updates descriptor defined in section Architecture is detailed in Figure 4. We have seen that a tester generates the data: $\langle \tau, \beta, \vec{\eta}, \nu, \varepsilon, \vec{\Phi}, \vec{\Upsilon}, \vec{\Theta}, \vec{\Psi}, d, \mathcal{S}\rangle$.

Algorithm 3: Updates Client

```
1  on clock do
2      for $url \in M$ do
3          d ← skew(url)+time();
4          send update_request(myID,url,d,ver,crt,$\mathcal{S}$);
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();
```



```
<versionInfo>
   <version>0.9.7</version>
   <date>20130116193137.658Z</date>
   <script>update</script>
   <build>1.7.0_02-b13</build>
 - <downloadables>
   - <downloadable>
       <url>http://my.site.edu/download/ddp2p_0.9.7.jar</url>
       <fileName>ddp2p_0.9.7.jar</fileName>
       <fileSize>4325224</fileSize>
       <digest>gJRDdPkosx+unGGYDdwRsy9oK20SDUTJ/XeEgu2nosQ=</digest>
     </downloadable>
     ....
   </downloadables>
 - <QOTD>
   - <testDef>
       <ref>0</ref>
     - <qualityStructure>
         <quality>Security</quality>
         <quality>attack</quality>
         <quality>buffer_overflow</quality>
       </qualityStructure>
       <desc>resistance to buffer overflow attacks</desc>
     </testDef>
     ....
   </QOTD>
 - <testers>
   - <testerInfo>
       <name>Allen Bond</name>
       <digestPK>SHA-1:uZ36bWcATfM4t1Gm7+tlu6KJz8g=</digestPK>
     - <QOTD>
         <testDef> .... </testDef>
         ....
       </QOTD>
     - <tests>
       - <test>
           <qualityRef>0</qualityRef>
           <QoT>0.7</QoT>
           <RoT>0.75</RoT>
         </test>
         .....
       </tests>
       <signature>dABIqIfMwATTZJzm...</signature>
     </testerInfo>
   </testers>
</versionInfo>
```

Figure 4: Updates Descriptor

This data can be encoded in the updates 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.

In Figure 4, the tester identifier τ appears within the tag: `<digestPK>`. The information about files $\vec{\eta}$ is indicated by `<downloadables>`. The quality definitions $\vec{\Phi}$ are specified in element `<QOTD>`, each quality being described in a sub-element `<testDef>`. The data from a tester is included in the tags `<testerInfo>` in the section `<testers>` of the descriptor. The entries of the quality of tests vector $\vec{\Theta}$ appear in the sub-elements `<QoT>` of the corresponding entry `<test>`. The entries of the result of tests vector $\vec{\Psi}$ appear in the corresponding sub-elements `<RoT>`, their index into the test vector being specified by the `<ref>` element. The additional quality definitions $\vec{\Upsilon}$ of this tester appear in the sub-element `<QOTD>` of `<testerInfo>`. The release date d is in element `<date>` and the signature $\mathcal{S}$ in the sub-element `<signature>` of the `<testerInfo>`.

The updates descriptor provided by mirrors as answer to update request has the same structure but can contain multiple `<testerInfo>` elements as sub-elements of the `<testers>` element.

Reference Implementation

Here we describe issues solved in our reference implementation of the proposed automatic updates system for the open source peer-to-peer DDP2P system which is implemented mainly in Java.

Distribution preparation: While the Java JDK7 compiler implements a *binary builder*, this does not hold for the corresponding `jar` archiever 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 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).

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 (Ayala 2004). It is a set of PHP classes used to generate 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 in Figure 2 and 3. Received updates are handled according to Algorithm 1 for automatic installation of updates.

The Decentralized Tester

As future work, we plan to investigate mechanisms for integrating recommendation systems technology into our framework to help building trust into testers. Similarly, a mechanism for reporting failures from the client to the mirrors, testers and developers can be used to improve testing procedure. We want to investigate the possibility of implementing a *virtual decentralized tester* based on the bug reports coming from peer users. Users can exchange information about bug reports and can vote on the quality of releases, updates as well as on the quality of various properties claimed for each release. This vote can refer to a RoT^q value assigned by a tester to a quality q. An alternative is for each user i to vote each quality q by providing their RoT_i^q and QoT_i^q values, and to aggregate the RoT_i^q values as a weighted sum, where the QoT_i^q weight is additionally weighted by the trust Φ_i in the user i providing it (Equation 1).

$$Rot^q = \frac{\sum_i RoT_i^q * QoT_i^q * \Phi_i}{\sum_i QoT_i^q * \Phi_i} \qquad (1)$$

Such votes can be aggregated by a server, or can be exchanged using a P2P platform to achieve the *fully decentralized tester*.

Analysis

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 resouce 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).

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).

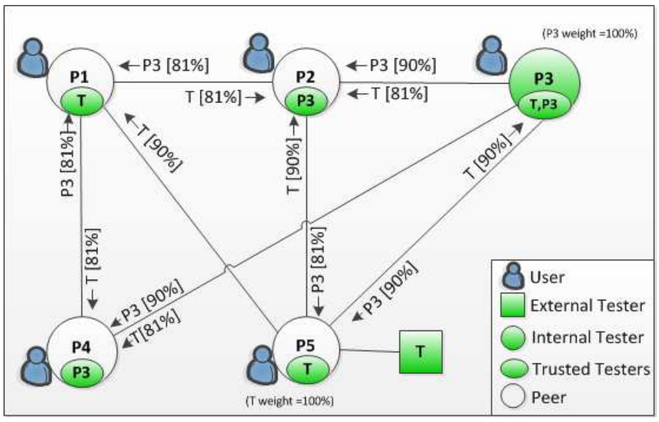


Figure 5: Overall Architecture of the Recommender System

Principles 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,

Definition 1 (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).

Definition 2 (Proximity) *Preference should be given to testers that are close to the user (geographically or in terms of some social network), as a heuristic to improve decentralization.*

Recommender Systems for testers in P2P applications

For P2P applications, such as DDP2P, there exists an intrinsic social network as defined by the connections of each peer (or constituent). In one such scheme, testers used by a peer are recommended to neighboring peers. Each tester being associated with a weigh (trust coefficient), this weight can decrease with each level of forwarding (using an amortization coefficient). 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. 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. 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 trust coefficient is amortized with 90% for each new link in the chain of recommendation, the obtained recommendation in a P2P application is shown in Figure 5. As shown, there are six peers (P1,..P6) that use two testers: P3 and T. P3 is a peer that is also a tester, called in the following *internal tester* and T is not part of the P2P network, being referred to as *external tester*. The user of P3 is using herself as a trusted tester and she has started giving herself a 100% as weight. P5 introduces and uses the external tester T, whom she also assigns a weight of 100%. Both P3 and P5 pass their selected testers information to neighboring peers. In Figure 5, P3 announces herself to her neighbors peers P2, P4 and P5 which see her recommended with the weight 90% (0.9*100%). Also, P5 recommended T to her neighbors peers P1, P2 and P3 which see the weight 90%. Based on these recommendations, P2 and P4 have decided to use P3 as trusted tester and forward P3's information to P1 which see the associated weight 81% (0.9 * 90%). In addition, P1 has decided to use T as a trusted tester (P1 had the choice to use P3[81%] or T[90%] or both). However, P3 has decided to use T as trusted tester beside herself.

Conclusions

An integrated framework for *development, testing, releasing and distribution of automatic updates* is proposed for the case of *open source software* maintained by *volunteers*. Close source software and open source software managed by commercial entities are controlled by a central authority that can be trusted and made responsible for low quality updates or infiltration of malware in updates. However, no such protection was so far available for volunteer-based open source software.

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. 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).

References

Adi, W.; Al-Qayedi, A.; Negm, K.; Mabrouk, A.; and Musa, S. 2004. Secured mobile device software update over ip networks. In *SoutheastCon*, 271 – 274.

Antwerp, M., and Madey, G. 2008. Advances in the sourceforge research data archive. In *Int. Conf. on Open Source Systems*.

Ayala, D. 2004. nusoap-web services toolkit for php. *Novembro, http://dietrich. ganx4. com/nusoap*.

Bellissimo, A.; Burgess, J.; and Fu, K. 2006. Secure software update:disappointments and new challenges. In *In USENIX Workshop on Hot Topics in Security (HotSec*, 37–38.

Bertino, E.; Correndo, G.; Ferrari, E.; and Mlla, G. 2003. An infrastructure for managing secure update operations on xml data. In *SACMAT*.

Bonnington, C. 2012. Recent software update is bricking some apple tvs. `http://www.wired.com/gadgetlab/2012/11/apple-tv-problems/`.

Cadar, C., and Hosek, P. 2012. Multi-version software updates. In *Hot Topics in Software Upgrades (HotSWUp), 2012 Fourth Workshop on*, 36 –40.

Cappos, J.; Samuel, J.; Baker, S.; and Hartman, J. H. 2008. A look in the mirror: attacks on package managers. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, 565–574. New York, NY, USA: ACM.

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

Dolev, D., and Yao, A. C. 1981. On the security of public key protocols. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, SFCS '81, 350–357. Washington, DC, USA: IEEE Computer Society.

Dromey, R. 1996. Cornering the chimera [software quality]. *IEEE Software* 13(1):33 –43.

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

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

Myers, M.; Ankney, R.; Malpani, A.; Galperin, S.; and Adams, C. 1999. X.509 internet public key infrastructure online certificate status protocol - OCSP. *IETF Network Working Group Request for Comments* 2560.

Nichols, D. M., and Twidale, M. B. 2003. The usability of open source software. In *First Monday, volume 8, number 1*.

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

Samuel, J.; Mathewson, N.; Cappos, J.; and Dingledine, R. 2010. Survivable key compromise in software update systems. In *CCS'10*, 61.