OSS Quality Assessment Trustworthiness[1]
(Working Paper)

Jean-Marc Seigneur

Abstract. Open Source Software (OSS) projects have the unique opportunity to reach an unprecedented level of software quality by tapping into its community and collaborative power. However, the community process of collaborative software Quality Assessment (QA) may not reach its full potential or worse be easily jeopardised by malevolent entities because there is a lack of protection mechanisms, easy-to-use enabling mechanisms and clear incentives. We propose such mechanisms as part of a decentralised collaborative test and QA framework centred on the OSS actors.

1 Introduction

A major asset of OSS projects lies in its collaborative and community values [1]. On one hand, it has been argued that the more OSS actors participate, the higher software quality is reached: Raymond argued that “given enough eyeballs, all bugs are shallow” [2]. On the other hand, recent investigations have claimed that few actors review OSS code and that it may not be enough to sustain high quality in OSS projects: “the vast majority of 'eyeballs' apparently do not exist” [3]. As the collaborative and community values are deeply anchored in the OSS ecosystem [1], the root of the problem seems clearly related to a lack of enabling technical mechanisms. The OSS community is present and is wiling to contribute but the tools to contribute are missing or too cumbersome to use. A few repositories aggregating OSS datasets and dashboards (Web portals presenting the information) have recently emerged but there is a lack of automated exchange and interoperability: the first workshop trying to harmonise them was run in June 2006 [4].

Fortunately, with the advances in Web technologies, sometimes called Web 2.0, it becomes easier and easier to build tools to network the community. The EU-funded EDOS project [5, 6] is building a formal information model called the Project Management Interface (PMI) [7] describing OSS artefacts, for example, Actor, Platform or Maintainer, and OSS activities, for example, SubmitPatch

[1] A revised version of this paper will be presented and published in the proceedings of SECOVAL 2007, Third International Workshop on the Value of Security through Collaboration, part of SECURECOMM'07, September 17, Nice, France.

activity and SubmitTestReport activity. In EDOS, a number of tools are being deployed to gather and distribute information about OSS projects in a peer-to-peer way reusing Web 2.0 building blocks such as XML, RSS feeds and REST/Web services [8].

For example, a Quality Assessment (QA) Web portal, called the EDOS dashboard, has been put in place to easily inform OSS actors of the quality of OSS projects computed based on information reported by other OSS actors who can deploy user-friendly tools on their platform to test and report specific OSS projects in an easy way.

Thus, the actors and their platform(s) play an important role in the EDOS-powered community process of collaborative OSS quality assessment and improvement. However, relying on external information submitted from an open ecosystem, decentralised by nature and populated by possibly competing actors, introduces a number of trust issues: competing actors (including actors outside of the OSS community) may try to submit false reports about the quality of the projects of their competitors; cheating is facilitated when actors can come and go without the feasibility of a centralised authentication service; and in the QA application domain, "trust in the accuracy of any test data [...] depend[s] on [...] trust in the providence of the testers" [9]. It seems fair to assume that in a small project all the OSS QA team actors may know face-to-face all the involved testers. However, it may not be possible for bigger projects because the OSS ecosystem is an open environment where actors and their digital identities can come and go. Nevertheless, due to the need of more actors carrying out test tasks, as long as the testers are trustworthy, they are welcome to contribute to the project quality improvement process. Some tests may have to be run on foreign willing peers by not-so-well-known testers and the test result needs to be adjusted based on who carried out the test and on which platform. It is especially relevant in the EDOS project that relies on a peer-to-peer storage layer for OSS information. There is the need for security/trust metrics to select the most trustworthy peers for efficient and safe distribution of QA information.

The remaining of this document proposes a framework to mitigate the above mentioned trust issues in collaborative OSS QA. The first step in this direction is to define how we can model the peers, their platform, their state and their actors. Then, Section 3 details how we can recognise a requesting or answering platform, which is an important aspect for trust. Section 4 describes how we take into account the actor aspect in the trust model. Finally, we draw our conclusion in Section 5.

2 Trust in the Platform Configuration

The first abstraction that we make is that a peer is a platform.

We assume that a platform has a number of actors, a software configuration and a hardware configuration. The hardware configuration seems to be the easiest to model since hardware is rather fixed compared to software. However, even the Trusted Platform Module (TPM) alliance [10] does not provide a clear model: it merely specifies that the hardware configuration schema is up to the TPM manufacturer after giving a few examples of what hardware could be measured, for example, the type of the memory of the machine or the CPU type. In addition, the TPM manufacturers do not seem to have publicly released their schema yet. It is the reason that our model states that a hardware configuration is an immutable composition of hardware elements, meaning that if a hardware element is added, removed or re-configured a new hardware configuration must be created. Anyway, it is not uncommon that some software works with some hardware and does not work with some special hardware. It is the reason that hardware is especially important in software QA.

Concerning software configuration in the literature, as underlined by Fielding in his PhD thesis [8] in 2000, "in spite of the interest in software architecture as a field of research, there is little agreement among researchers as to what exactly should be included in the definition of [software] architecture". We build on Fielding's model for software architecture whose model is summarised below:

- "A software architecture is an abstraction of the run-time elements of a software system during some phase of its operation. A system may be composed of many levels of abstraction and many phases of operation, each with its own software architecture." [...] "A software architecture is defined by a configuration of architectural elements − components, connectors, and data − constrained in their relationships in order to achieve a desired set of architectural properties. [...] In addition to levels of architecture, a software system will often have multiple operational phases, such as start-up, initialization, normal processing, re-initialization, and shut-down. [...]
- A component is an abstract unit of software instructions and internal state that provides a transformation of data via its interface. [...]
- A connector is an abstract mechanism that mediates communication, coordination, or cooperation among components. [...]
- A datum is an element of information that is transferred from a component, or received by a component, via a connector. [...]
- A configuration is the structure of architectural relationships among components, connectors, and data during a period of system run-time. [...]
- both the functional properties achieved by the system and non-functional properties, such as relative ease of evolution, reusability of components, efficiency, and dynamic extensibility, often referred to as quality attributes [are considered]" [8]

In our testing and QA case, we focus on the non-functional requirements of security and trust. According to Fielding, we need to find the correct level of abstraction of the run-time elements of a software system, which underlines that configurations are dynamic. From a trust and security point of view, it seems important to keep the history of the different platform configurations to be able to retrieve which configuration change created a new bug/vulnerability or which configuration fails a specific test as well as to evaluate the risk of a platform.

Our model also includes actors and hardware. The originality of our model compared to the other platform integrity models surveyed below is that we also include the users of the platform in the loop.

Maruyama et al. empahised that “it is important to understand that the integrity of the platform, including the hardware, the operating system, the shared libraries, and the all the related configuration files, is essential for the integrity of the application component because ultimately the component's behavior relies on the platform” [12]. They use a modified version of the GRUB kernel loader to chain Platform Configuration Registers (PCRs) in the Trusted Platform Module (TPM) of the platform. The measured integrity values can be reported to a requesting machine at time of signature verification using Java once their new signature algorithm is plugged into the Java Crypto Environment (JCE). The attestation public key corresponds to a pseudonym of the platform endorsement key and certified by a known privacy Certification Authority (CA). Again, the reported PCRs should correspond to values that are known to correspond to trusted elements. The signature is not validated if the PCRs do not correspond to trusted values. If all the platform elements would be measured, it would lead to a combinatorial number of PCR values. It is the reason that they let the Operating System (OS) kernel take care of the integrity of less security-critical files and resources. They used tools from the security enhanced Linux [13]. A lighter/weaker mechanism may be to monitor critical elements, for example, with Tripwire [14], Advance Detection Intrusion Environment (AIDE) [15] or File Alteration Monitor (FAM) [16], which is very efficient when /dev/imon is available because it receives notifications directly from the kernel rather than polling for information. It is worth noting that the integrity of the platform measured at bootstrap may not be valid if the OS has let a program modify the elements that are measured.

Zheng and Cofta [17] emphasise that “one disadvantage of the current [TPM] paradigm is that it does not provide a dynamic solution”: “trust relationship might break after a period of time”. Instead of using the simple network-expensive solution to periodically re-challenge the remote platform, they propose an extended TPM hardware, called the “root trust module” that monitors any platform change (hardware, software and their configurations) and notifies the requester if the trust conditions become not respected. Of course, it is supposed that external mechanisms cannot force the root trust module once both trustee and trustor have agreed about the trust condition (for example, not running an ftp server).

Jaeger et al. [18] have recently extended the TPM-based IMA Linux architecture [19] because previous TPM approaches only provide load-time guarantees, which do not accurately reflect runtime behaviours. Thanks to policies that specifies the elements that have to be measured, they are able to measure information flow integrity guarantees at runtime that can be verified by remote parties. As a side effect, the measurement step is more efficient because fewer elements have to be measured. However, as Maruyama et al. [12] concludes, further thorough investigations are needed to define which “pieces of the OS should be measured to ensure the overall integrity of the platform”.

Thus, from the above related work, it is clear that the dynamic aspect of the platform is important as depicted in the following Figure 1.

Ideally, assuming that no bug can crash/switch-off the platform, a *PeriodOfPlatformRuntime* should also be created and saved during the *Boot* and *Shutdown* states, similarly to the internal actions mentioned in the *Normal Processing* and *Re-initialisation* states.

Our model allows a remote platform to access TPM-guaranteed information, such as PCR values, if the platform is TPM enabled. However, as we have seen above, there are only a few Linux distributions that are TPM-enabled. Therefore, our model also provides software and hardware configuration that has been collected with standard tools such as rpm -qa, which gives a complete list of all installed rpm packages on a given platform, for Linux software configuration and lshw (Hardware Lister) [20] for hardware configuration, reporting, for example, exact memory configuration, firmware version, mainboard configuration, CPU version and speed, cache configuration, bus speed, etc.

Concerning security information (theoretically) enforced by a specific OS distribution, such as Mandriva's security level configuration: 5 different levels can be configured from weak security to paranoid security, this type of security information can be reported as the configuration parameters of the software element state of the software element representing the OS.

The event called *Major change detected* in the center of the figure means that, for example, one of the following changes has occurred and according to the current policy a new period of platform configuration must be created:

- A new package has been installed;
- A new actor has logged in;
- An attack or a vulnerability has been detected on one of the components of the platform ;
- A risky component (for example, one opening a network port) has been either started, stopped or its configuration has been changed;
- An important hardware modification has been detected.

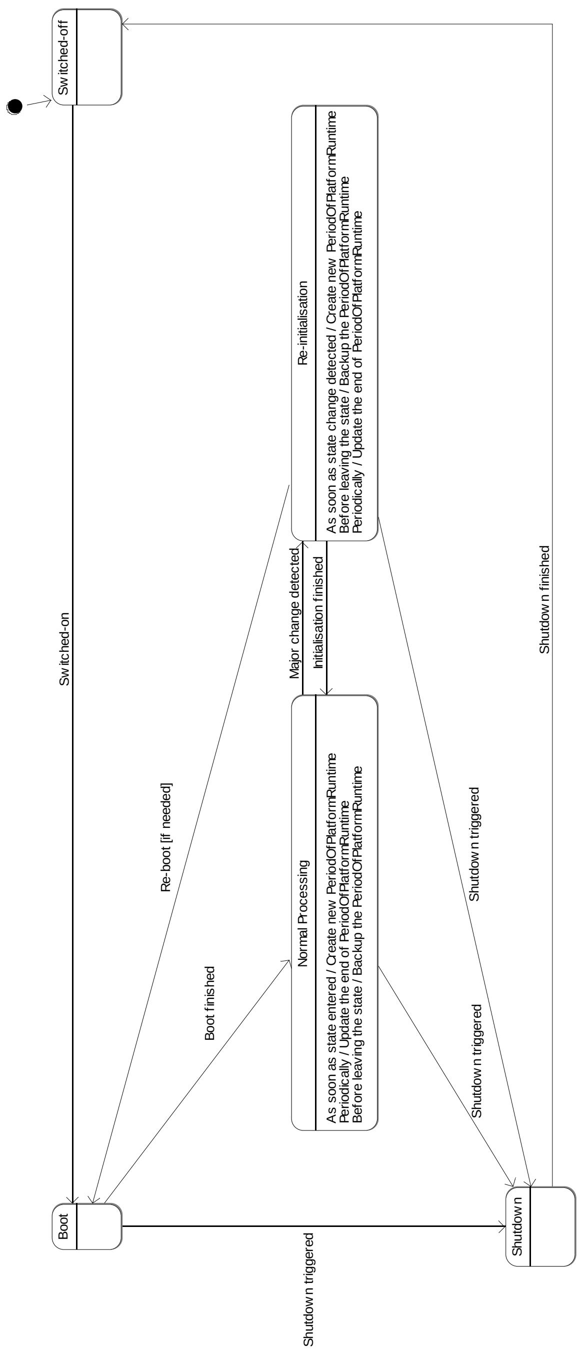
Switched-off
Switched-on
Boot
Re-boot [if needed]
Boot finished
Normal Processing
As soon as state entered / Create new PeriodOfPlatformRuntime
Periodically / Update the end of PeriodOfPlatformRuntime
Before leaving the state / Backup the PeriodOfPlatformRuntime
Major change detected
Initialisation finished
Re-initialisation
As soon as state change detected / Create new PeriodOfPlatformRuntime
Before leaving the state / Backup the PeriodOfPlatformRuntime
Periodically / Update the end of PeriodOfPlatformRuntime
Shutdown triggered
Shutdown triggered
Shutdown triggered
Shutdown
Shutdown finished

Fig. 2. Platform State Diagram.

3 Trust in the Recognition of the Platform

As said above, although a TPM hardware solution seems one of the strongest approach for platform identification, few platforms are already TPM-aware. However, it still seems possible to identify a platform based on a combination of means not protected by a TPM.

For example, Microsoft has put in place a scheme to uniquely identity platforms based on a combination of hardware configuration hashes and globally unique identifiers sent to the target platforms. More precisely, Microsoft Genuine Advantage program consists of the actor having to download GenuineCheck.exe and run it. Thereafter if the platform and its OS are supposed to be genuine, the actor receives a code that can be used to download upgrades and other goodies such as IE 7. During the validation, at time of writing [21], the following configuration information is collected: the computer make and model; the version information for the OS and software using Genuine Advantage; the region and language setting; a Globally Unique Identifier (GUID) assigned to the platform by the tools; the product ID and product key; the BIOS name, revision number and revision date; the volume serial number and the Office product key (if validating Office). The validation tools do not collect the actor's name, address, e-mail address, or any other information that Microsoft could use to identify or contact the actor. In addition to the above configuration information, the result of the validation check and whether or not the installation was successful are transferred.

Because there are a number of more or less strong means to identify a platform, we introduce a *RecognitionManager* that is in charge of recognising a platform based on the recognition clues that the different means provide: the more recognition clues are validated, the higher confidence we can have in the platform identification. A fully fledged discussion of how entity recognition differs from authentication can be found in [22].

The first recognition clue that we propose to use to recognise a platform is the signature by an asymmetric cryptographic private key known to be owned by the platform under scrutiny. A private key hash can also be considered as a GUID as the above Microsoft GUID although its value is not controlled by anybody but randomness. It may be quite vain to rely on security through obscurity by somehow binding the GUID with the platform configuration because it may be quite straightforward to spoof the properties that are used to generate the GUID. We reuse the EDOS [6, 11] peer-to-peer (P2P) infrastructure API that provides key creation, certification, verification and revocation. An extra requirement that we put on platform key creation is that only root/administrator actors are allowed to create platform keys in order to avoid having simple users compromising the trustworthiness of a platform that they do not own.

However, the latter requirement could limit the number of actors being able to contribute test activities. It is the reason that ideally an administrator would start a service available to other users with access to the platform private key for signing test reports and the platform current configuration and history. The latter service should not need to be run as a root process to avoid major security issues if the service becomes compromised.

The sequence diagram depicted in Figure 2 indicates the steps that should be followed to use this first type of recognition clue.

In case the platform is TPM-aware, the step to create the new key pair would be done by the TPM rather than not protected software. The TPM would create an Attestation Identity Key (AIK) that is certified by a Privacy CA known by the different parties involved in QA. As an AIK can only be created by the TPM owner, such a key provides good evidence that the actor has root privileges on the platform. In case the platform is not TPM-enabled, we assume that the EDOS P2P infrastructure provides a platform key certification service that challenges the platform to find out whether or not the current requester has access to root privileges, for example, a simple network socket must be opened on a low port only accessible to administrators and reply to a specific challenge. First, the service certification queries the P2P storage to find out if there is already a key bound to the given platform configuration; this step of recognising the platform may involve using a *RecognitionManager* owned by the certification service. Then, in case of no previous key bound to the platform and a successful challenge, the service creates a certificate stating that the public key corresponds to a new platform key and publishes the new key along its certificates and configuration in the standard P2P storage way. Optionally, the platform may start the above mentioned local testing service for potential local users or even remote testers. If the local owner is prepared to allow remote testers to carry out tests locally, the P2P storage provides an easy way to retrieve which platform configurations are available.

It is worth mentioning that the weakest root privilege challenge/response and previous platform key binding verification mechanisms correspond to not doing anything and simply returning true at each request.

There are other means that could be used as recognition clues. For example, the requester previous platform history may be known by the verifying platform and the requester platform may be asked to specify which configuration it had in the past during a specific period of time. Another means may be to challenge/response the requester platform based on its IP address. If the requester platform would be able to reply a specific signed nonce that was sent to the specific IP address, then it means that at least the confidence in the recognition of the requester platform is as good as the confidence that can be put in DNS-based security. The sequence diagram depicted in Figure 3 represents the basic steps that the *RecognitionManagers* of the requester platform and the receiver of the test report would have to carry out. It should be possible to add different recognition means/schemes to the *RecognitionManager* and use them to compute an overall level of confidence in recognition in the platform that sent the test report or defect. The test report may either include the level of confidence in recognition or just be discarded if the level of recognition is too low, meaning that a platform has been spoofed.

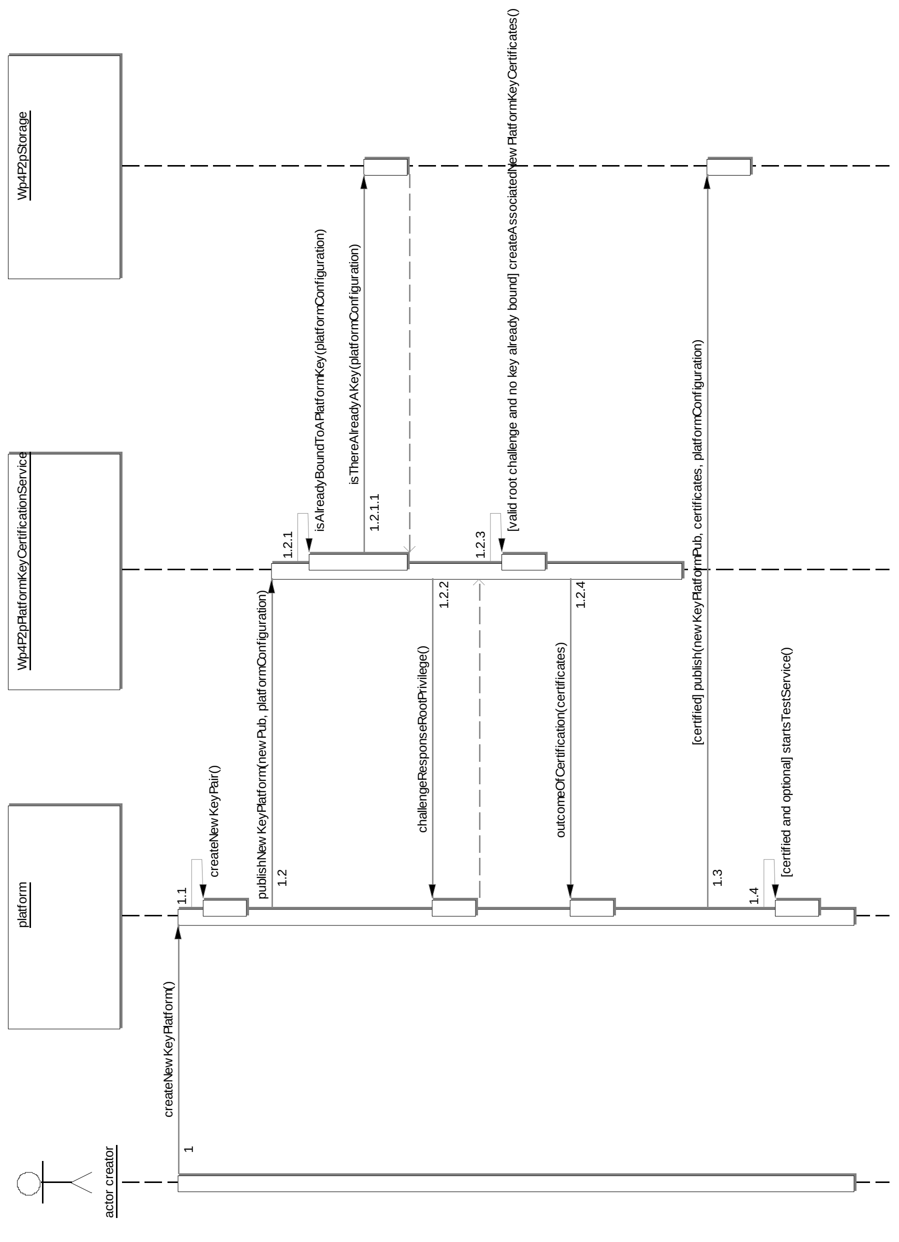
actor creator
platform
Wp4P2pPlatformKeyCertificationService
Wp4P2pStorage
1
createNewKeyPlatform()
1.1
createNewKeyPair()
1.2
publishNewKeyPlatform(newPub, platformConfiguration)
1.2.1
isAlreadyBoundToAPlatformKey(platformConfiguration)
1.2.1.1
isThereAlreadyAKey(platformConfiguration)
1.2.2
challengeResponseRootPrivilege()
1.2.3
[valid root challenge and no key already bound] createAssociatedNewPlatformKeyCertificates()
1.2.4
outcomeOfCertification(certificates)
1.3
[certified] publish(newKeyPlatformPub, certificates, platformConfiguration)
1.4
[certified and optional] startsTestService()

Fig. 2. Platform Key Certification Sequence Diagram.

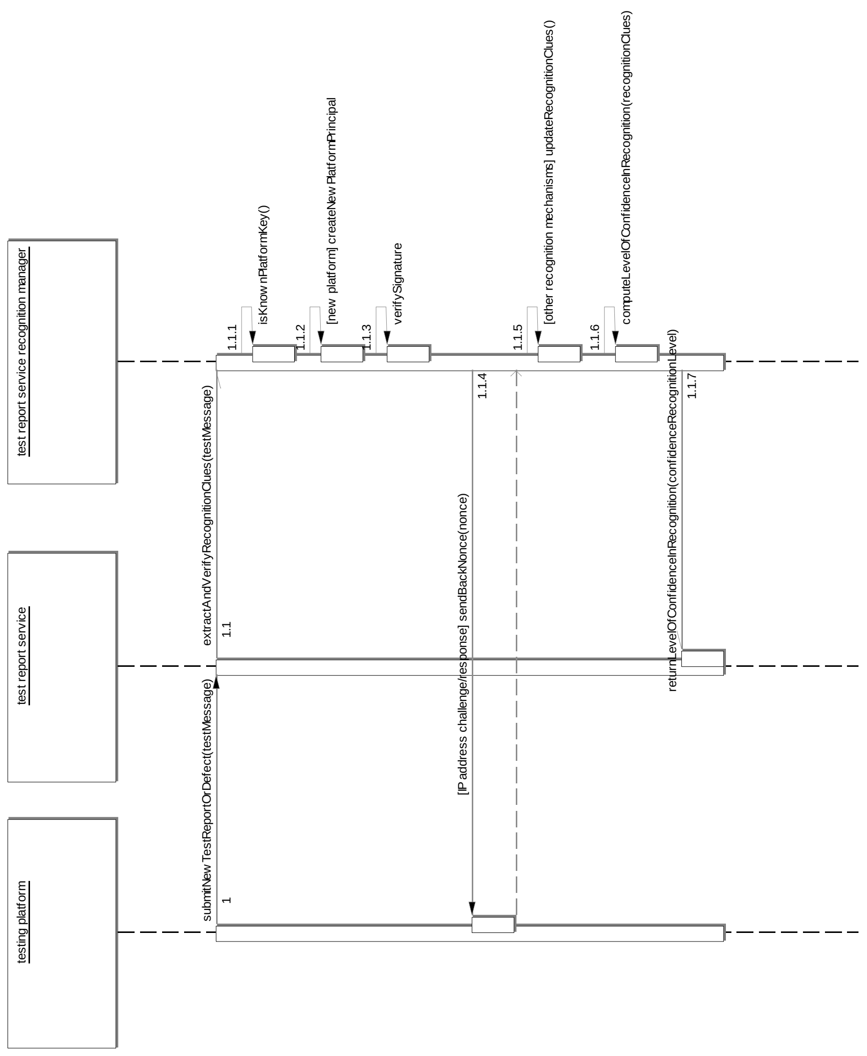
test report service recognition manager
test report service
testing platform
1
submitNewTestReportOrDefect(testMessage)
1.1
extractAndVerifyRecognitionClues(testMessage)
1.1.1
isKnownPlatformKey()
1.1.2
[new platform] createNewPlatformPrincipal
1.1.3
verifySignature
1.1.4
[IP address challenge/response] sendBackNonce(nonce)
1.1.5
[other recognition mechanisms] updateRecognitionClues()
1.1.6
computeLevelOfConfidenceInRecognition(recognitionClues)
1.1.7
returnLevelOfConfidenceInRecognition(confidenceRecognitionLevel)

Fig. 3. Recognition Manager Sequence Diagram.

As it is also important to know who the tester that sent the report or defect is to estimate how trustworthy the information is, the *RecognitionManager* can also recognise the signature of the tester. In fact, we assume that the test report or defect is signed by both platform private key and tester's private key, which corresponds to the standard private key that the EDOS P2P infrastructure manages for each actor.

4 Trust in the Tester Quality

If we assume that there are many more trustworthy testers, tests and platforms than untrustworthy ones, the correlation of all the test reports and defects should allow us to detect untrustworthy tests and platforms. For example, tests that always fail although the software as such would work; or compromised platforms sending false reports or defects. More precisely, if one out of fifty platforms with the same hardware and software configuration reports that many different tests crashed the platform whereas all the other platforms report that all theses tests passed, there is a chance that the platform itself is buggy and thus should be considered untrustworthy.

Another reason for deviant reports may be that the tester is untrustworthy, for example, because she/he is involved in a competing project or not very talented for testing.

Recent work on test quality has emphasised the importance of tester quality: “your trust in the accuracy of any test data will depend on your trust in the providence of the testers / quality of the testers” [9]. Although this related work is a first step towards taking into account the importance of tester quality, they only use fixed, manually configured, subjective trust levels about testers/developers quality, for example, the average testing quality as estimated by the manager of the QA. We detail below how the EDOS QA information model and processes allow us to easily collect and distribute tester quality and trust objective evidence.

There are different aspects that can be taken into account with regard to the testers quality and trustworthiness. To start with, there is a difference between an end-user actor who is just willing to run a specific test on his/her platform and the tester actor who has designed this test in the first place. Secondly, on one hand, a tester should be rewarded when a test carried out before the software release detects bugs, especially this trust context. By test report trustworthiness, we mean the number of times that the tester has reported the same test outcome as the majority of testers with similar platform configuration. The context test contribution captures the number of times the tester has contributed and spent time for the OSS community on QA tasks. The activity diagram of Figure 5 depicts one way to update these counters.

Depending on the criticality of the bug found by an end-user actor when using the software, we may modulate the test quality trustworthiness counters by the criticality of the bug. For example, given:

- a tester *u;*
- *N* independent tests carried out by *u;*
- *crit*, the criticality between [0,1] of a bug found to be linked to a specific test

testqualitytrustworthiness(u), an estimation of the trustworthiness of the test quality of the tester *u* may be:

testqualitytrustworthiness(u) = (Sum of (1-crit) for all tests carried out by u)/N

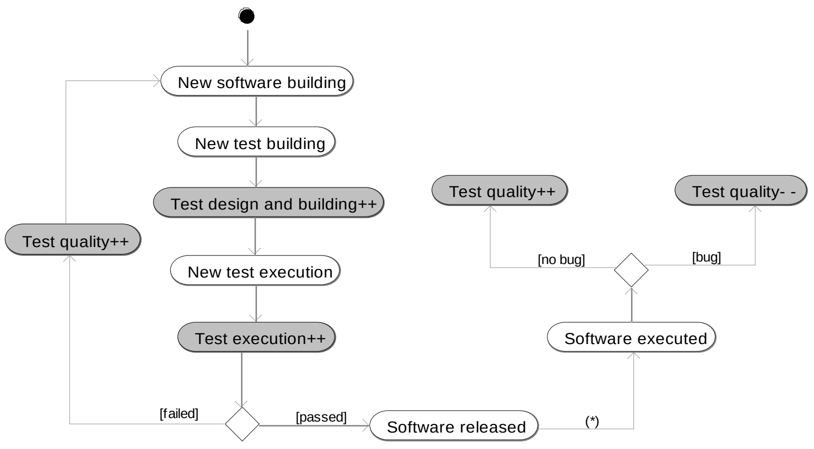

Fig. 4. Tester Trustworthiness Update.

5 Conclusion

We have presented an open source software quality assessment framework that can dynamically take into account software and hardware configuration trustworthiness as well as the trustworthiness of the testers.

Such a framework may be used in OSS QA portals such as the EDOS quality assessment portal. In the future when trusted platform modules are available, it will be become even easier to recognise the different involved parties and the QA results will gain in trustworthiness.

References

[1] Stewart, K. J. and Gosain S. (2001) An Exploratory Study of Ideology and Trust in Open Source Development Groups, in *International Conference on Information Systems*.
[2] Raymond, E.S. (1997) The Cathedral and the Bazaar.
[3] Schach, S.R.(2004) Colloquium Presentation.
[4] González-Barahona, J. M.; Conklin, M. and Robles, G. (2006) Public Data about Software Development, in *International Conference on Open Source Software*.
[5] Abiteboul, S.; Leroy X.; Vrdoljak B. et al. (2005) EDOS: Environment for the Development and Distribution of Open Source Software, in *International Conference on Open Source Software*.
[6] Environment for the development and Distribution of Open Source software (EDOS), http://www.edos-project.org/.
[7] Pawlak, M. (2005) Project Management Interface (PMI), ASG Technical Report, 2005.
[8] Fielding, R. T. (2000) Architectural Styles and the Design of Network-based Software Architectures, PhD Thesis, University of California, Irvine.
[9] Fenton, N. and Neil, M. (2004) Combining Evidence in Risk Analysis using Bayesian Networks, Agena Technical Report.

[10] Trusted Computing Group, https://www.trustedcomputinggroup.org/.
[11] Seigneur, J.-M. (2006) Security Evaluation of Free/Open Source Software Powered by a Peer-to-Peer Ecosystem, Workshop on Evaluation Frameworks for Open Source Software, OSS International Conference.
[12] Maruyama, H.; Nakamura, T.; Munetoh, S. et al. (2003) Linux with TCPA Integrity Measurement, IBM Technical Report.
[13] Security Enhanced Linux, http://www.nsa.gov/selinux/.
[14] Tripwire, http://www.tripwire.com/.
[15] Advanced Intrusion Detection Environment, http://sourceforge.net/projects/aide.
[16] File Alteration Monitor (FAM), http://savannah.nongnu.org/projects/fam/.
[17] Zheng, Y. and Cofta, P. (2006) A Mechanism for Trust Sustainability Among Trusted Computing Platforms, in *Trust and Privacy in Digital Business*. Springer.
[18] Jaeger, T.; Sailer, R. and Shankar, U. (2004) PRIMA: policy-reduced integrity measurement architecture, in *the Proceedings of the 11th Symposium on Access Control Models and Technologies*, ACM.
[19] Sailer, R. (2004) Integrity Measurement Architecture, IBM Technical Report.
[20] Hardware Lister (lshw), http://ezix.org/project/wiki/HardwareLiSter.
[21] Genuine Microsoft software FAQ, http://www.microsoft.com/genuine/downloads/faq.aspx.
[22] Seigneur J.-M. (2005) Trust, Security and Privacy in Global Computing, Trinity College Dublin, PhD Thesis, Technical Report.