Managing Software Uninstall with Negative Trust

Giuseppe Primiero, Jaap Boender

Managing software uninstall with negative trust

Giuseppe Primiero and Jaap Boender

Department of Computer Science
Middlesex University
United Kingdom
G.Primiero@mdx.ac.uk
J.Boender@mdx.ac.uk

Abstract. A problematic aspect of software management systems in view of integrity preservation is the handling, approval, tracking and eventual execution of change requests. In the context of the relation between clients and repositories, trust can help identifying all packages required by the intended installation. Negative trust, in turn, can be used to approach the complementary problem induced by removing packages. In this paper we offer a logic for negative trust which allows to identify admissible and no-longer admissible software packages in the current installation profile in view of uninstall processes. We provide a simple working example and the system is formally verified using the Coq theorem prover.

1 Introduction

Software management configuration is among the most pervasive problems in modern personal computing, with complications caused by multiplication of users, required support for several software versions releases, increasing customization options and the need of coordination across distributed systems. One specific aspect of configuration management activities is change management, i.e. the handling, approval and tracking of change requests, with the aim of preserving the integrity of the system.

Consider the following example. A user interacts with a software package system to install or remove applications. The set of packages installed on a machine is called the installation profile of that machine. A valid installation profile is one which meets all the dependencies and conflicts clauses of all the packages installed and such that it satisfies sufficient dependencies for any desired package to be executed. Assume the current installation profile contains: two packages ϕ_1, ϕ_2 from the main repository; one package ψ_1 from the free repository; and one package ξ_1 from the non-free repository. Assume moreover that ψ_1 depends on ϕ_1 and from ϕ_2, while ξ_1 depends on ψ_1. Consider now the situation where the user wishes to prevent installation of a given additional package ψ_n from the free repository, while wishing to install a package ξ_2 from the non-free repository: which other packages is she safe in installing? and which ones does she need to remove in order to avoid conflicts in the new installation? Determining

these consistency relations between packages in a given installation is essential for system stability, but also to prevent the possibility of security threats in critical systems.

In [15], the problem of maintaining profile consistency and system integrity in view of uninstall processes is presented in the following terms:

Definition 1. *(**Uninstall Problem**). Given a new package ϕ to install, determine the minimal number of packages (possibly none) that must be removed from the system in order to make ϕ installable.*

This means identifying and removing all packages that are in conflict with the intended installation and its dependencies. This version of the problem can be complemented by that of identifying packages that depend on an undesired one.

In this context, trust can be used to characterize the relations between clients, software packages (including their dependencies) and repositories during the installation process. A software package in conflict with the current installation profile can not be trusted under it and hence not installed; if already installed, trust needs to be removed. Hence, dealing with such processes requires an explicit treatment of *negative trust*. Here and in the following the term *untrust* is used as neutral for 'negative trust' with respect to its derivatives *mistrust* and *distrust*: the former expresses trust removal, the latter trust denial. It should be noted that we refer to negative trust in the sense of being obtained through logical negation, as opposed to other quantitative approaches, where negative numbers are used. In [12] a natural deduction calculus is formulated which offers a proof-theoretical semantics for both notions. On this basis, we adapt here the Uninstall Problem from Definition 1 to the two semantics of untrust:

- A user identifies a package ϕ which generates conflict with a desired installation; to preserve profile consistency, ϕ is *distrusted* while the set of packages not depending on ϕ remain installable;
- A user identifies a package ϕ to be installed but in conflict with the current profile; to preserve profile consistency the packages $\psi_i, \ldots, \psi_n$ in the installation profile in conflict with ϕ are *mistrusted*.

The Uninstall Problem from Definition 1 can then be reformulated accordingly in the two variants:

Definition 2. *(**Distrusted Uninstall Problem**) Given a package ϕ that should not be installed, determine which other packages can be installed (i.e. that do not require ϕ).*

In this case, we are obviously interested in determining the maximal set of installable packages that do not conflict with ϕ.

Definition 3. *(**Mistrusted Uninstall Problem**) Given a package ϕ that should be installed, but which is in conflict with the current profile, determine which packages need to be uninstalled in order for ϕ to become installable.*

As in the approach from [15], we are interested here in determining the minimal set of packages inconsistent with ϕ that have to be removed from the installation profile.

In the present paper we provide a solution to these two problems in software management through their formalization in a logic for negative trust. In our model we use a trust function to allow access relations that presuppose consistency; in the current interpretation, trust (and hence of consistency checks) applies to software packages and conflicts are treated through negative trust. Note that the kind of inconsistencies we consider are not just those induced by technical requirements of the packages, but also by security issues. This formal strategy can help in offering a computable approach to trust management and in reducing risks related to installation profile inconsistency. The logic allows to reason about statements of the form:

> Installation profile Γ allows consistent installation of package ϕ and prevents installation of conflicting package ψ.

This approach is in the first place novel from a conceptual point of view, because software dependency satisfaction as trust management has not yet been largely investigated. Secondly, it is novel from a technical point of view, as proof-theoretic solutions and the possibility of implementation in theorem provers for automatic inconsistency checking have been neglected so far. In comparison with existing approaches for the resolution of inconsistent installations, our underlying logic allows a finer-grained approach than, for example, SAT-solvers.

The paper is structured as follows. In Section 2 we offer an overview of related works in the area of computational trust and software management. In Section 3 we introduce the system `(un)SecureND`, which provides the formal machinery for our analysis. In Section 4 the Distrusted Uninstall Problem is reformulated within our logic and its solution illustrated. In Section 5 the same is done for the Mistrusted Uninstall Problem. In Section 6 we present a simple scenario modelled by example derivations showing both cases at work. We conclude with some general remarks and a brief overview of future work.

2 Related Work

The present work sits at the intersection of the literature on software dependency management and computational trust. In this section we briefly overview related works in both areas and compare those to our approach and results.

In [5], we have offered a trust-based version of the optimization problem from [15], known as the *minimum install problem*, determining the optimal way to install a new package, where optimality is determined by an objective function to minimize the amount of dependencies satisfied such that it results in a valid installation profile. Trust is then used to guarantee that the minimal amount of dependencies for each newly installed package is satisfied by transitively accessed repositories. The complementary problem of maintaining profile consistency and

system integrity in view of uninstall processes can be similarly developed by applying the logic from [12] to the software management context.

In the context of software management, SAT solving appears as a promising approach for the development of efficient methods of dependency graph resolution. SAT technology has been used in [9] to validate dependencies and check installability of packages of specific Linux distribution. In Section 1 we have illustrated our current task as resolving two variants of the *Uninstall Problem* from [15]. In that work the Opium package-management tool is introduced, also based on pseudo-boolean solvers. Opium is complete with respect to solution finding and can optimize a user-defined function, e.g. to prefer smaller packages over larger ones. An implementation of Opium is available as the 0install solver.[1] A review of state-of-the-art package managers and their ability to keep up with evolution and their dependency solving abilities is offered in [1], with a proposal to treat dependency solving as a separate concern from other upgrade aspects. The upgrade problem is also considered in [2] to justify the design of a modular package manager. While we do not have an implementation of preferential settings based on user-choices, our installation profiles are defined according to a criterion of minimality for dependency satisfaction: this means that we construct installation profiles according to an ordered criterion of dependency satisfaction and package removal from a profile always proceeds to identify the minimal number of required packages. Also, in our approach we do not explicitly distinguish cases of upgrade as separate from installation of new packages: this is clearly a simplification, but the system can deal with upgrade with the more complex tactic of removing older versions and installing newer ones. The solvability of the decision problem related to software dependency management and its optimization are also considered in [3]. In the present paper our aim is to start an investigation in a proof-theoretical and trust-based approach to software dependency management, which so far has been neglected. We also hope to facilitate the introduction of automated theorem provers in the area, which can be beneficial in the checking process of intended installations in order to anticipate possible conflicts.

An associated but distinct issue is the *co-installability problem*: to quickly identify the components that can or cannot be installed together. It is related to boolean satisfiability and it is known to be algorithmically hard. It is shown to be especially complex for cases that include optimization by user preferences, where a combination of exact and approximate solving can help, [7]. In [16] a formally certified semantic method preserving graph-theoretic transformations is developed to associate to each concrete component repository a much smaller one with a simpler structure. One aspect of co-installability is that of reciprocal dependencies [4], which as mentioned more explicitly later is abstracted from in the present formulation. The *Mistrusted Uninstall Problem* formulated below replicates the intuition of the co-installability problem in the setting for external

[1] See `http://0install.net/solver.html`. An OCaml implementation is also available at `http://roscidus.com/blog/blog/2014/09/17/simplifying-the-solver-with-functors/`.

packages (and their dependencies) which are in explicit conflict with currently installed ones (and those they depend on). As for the latter work and the work presented in [1], our system enjoys a formal translation to a library for the Coq theorem prover,[2] with the aim of verifying its results. Our system seems also to be the only one among those in the area of software management that relies on the explicit formulation of a natural deduction calculus.

An essential characteristic of the method implemented in our system is that integrity checking on installation profiles is guaranteed through an explicit formulation of a trust access function on packages. The logic was first introduced in [13] and extended to deal with negative trust in [12]. Recently, research has started considering the advantages, implications and formal requirements needed to deal with the various aspects of negated trust, and in particular the different meanings that can be attached to mistrust and distrust, including the extension and limits of their transitivity and propagation protocols [11, 6, 10, 17]. Most current research ignores the difference between the procedural semantics of these two terms, possibly with the exception of [10], which presents mistrust as misplaced trust, untrust as little trust and distrust as no trust. This approach abstracts, though, from the reasons behind the attribution of these evaluations, in favour of a purely quantitative approach. Propagation for negative (first-order) trust is formulated in [8]. Our contribution relies on a strict distinction between *distrust* and *mistrust*: the former is intended as trust denied to packages coming from outside of the current installation profile in view of inconsistencies with currently installed ones; the latter is understood as trust revoked to installed packages, in view of desired new packages to be installed. These two cases have not been in general treated separately. Our approach formalises them in the context of uninstall operations, which as far as we are aware are entirely missing from the literature. Moreover, treating (un)install operations in terms of (un)trust allows us to integrate a consistency check performed over profiles that satisfy dependencies for the packages involved.

3 (un)SecureND

(un)SecureND is a natural deduction calculus defining trust, mistrust and distrust protocols introduced in [13] for the positive fragment and in [12] for the negation complete extension. We offer here a slightly modified version adapted for the software management problems at hand. In particular, the present version introduces a strict partial ordering on formulas to express package dependency; this is then lifted at the level of contexts to express rules for installation profile construction and finally imported at the level of repositories where the associated packages are located. In view of this order relation the system qualifies as a substructural logic, in that Weakening is constrained by a trust function, Contraction and especially Exchange by the order relation.

We start with introducing the language of our logic:

[2] The repository is available at https://github.com/gprimiero/SecureNDC.

Definition 4. *(**Syntax of** (un)SecureND)*

$\mathcal{S}^{\sim} := \{A < B < \ldots\}$
$\phi^S := a^S \mid \neg\phi_i^S \mid \phi_i^S \to \phi_j^S \mid \phi_i^S \wedge \phi_j^S \mid \phi_i^S \vee \phi_j^S \mid \bot \mid Read(\phi^S) \mid Write(\phi^S) \mid Trust(\phi^S)$
$\Gamma^S := \phi_i^S \mid \phi_i^S < \phi_j^S \mid \Gamma^S; \phi_j^S$

3.1 Repositories, packages and dependencies

$\mathcal{S}^{\sim}$ is the set of software repositories ordered by $<$ in view of dependencies between packages they contain, obtained below as lifting from package dependency. ϕ^S is a meta-variable for formulae, expressing software packages and their logical composition inductively defined by connectives, including operations to read (query), trust (consistency checking) and write (install). The language includes $\bot$ to express conflicts: we formulate $\neg\phi_i^A$ as an abbreviation for $\phi_i^A \to \bot$. Packages are typed by their origin in repositories: ϕ_i^S says that package ϕ_i can be retrieved from repository $S \in \mathcal{S}$. An installation profile Γ^S is the list of all packages sufficient to an access or execution operation; a profile is internally structured to reflect the dependency of packages through the partial order $<$ in $\mathcal{S}^{\sim}$. We allow extension of profiles by packages that are not dependent on previous ones, denoted by $\Gamma^S; \Gamma^{S'} = \{\phi_i^S < \ldots < \phi_n^S; \phi_{n+1}^{S'}\}$. This construction allows us to consider installation profiles that have all the sufficient conditions for the valid execution of a package, but can also be extended with additional packages. When such extension comes from the same repository, we use a comma: Γ^S, ϕ_i^S. The partial order allows for branching in the hierarchy, so that e.g. $\phi_1^S < \phi_2^S < \phi_3^S$ and $\phi_1^S < \phi_2^S < \phi_4^S$, i.e. packages ϕ_3^S, ϕ_4^S have both dependencies on ϕ_2^S and transitively on ϕ_1^S, but ϕ_3^S, ϕ_4^S could have no dependencies on each other.

Definition 5 (Judgements). *An* (un)SecureND*-judgement $\phi_i^A \vdash \psi_j^B$ says that a package ψ_j from repository B can be validly executed under a profile containing package ϕ_i from repository A.*

Definition 6 (Validity). *An* (un)SecureND*-judgement $\vdash \phi_i^A$ says that a package ϕ_i from repository A can be executed in any profile.*

We now generalise the dependency relation between packages $\phi_i^A < \psi_j^B$ at the level of repositories. A partial order relation $<$ over $\mathcal{S} \times \mathcal{S}$ intuitively expresses that dependencies are satisfied across repositories.

Definition 7. *$A < B$ iff $\exists \phi_i^A, \psi_j^B$ s.t. $\phi_i^A < \psi_j^B$ and $\neg\exists \phi_k^A, \psi_l^B$ s.t. $\psi_l^B < \phi_k^A$.*

By the first clause in Definition 7, $A < B$ means that some package in A satisfies a dependency for a package in B. By the second clause in Definition 7, our order relation abstracts from the issue of reciprocal dependencies. As noted in [4], two packages that mutually depend on each other will either be installed together, or not installed at all. They can therefore be considered as a single package for dependency resolution purposes. Rules from Figure 1 define installation profiles construction from packages dependencies. Here we use the extra-theoretical

$$\frac{}{\{\} : profile}\ \text{Empty Profile} \qquad \frac{\vdash \phi_i^A}{\phi_i^A : profile}\ \text{Package Insertion}$$

$$\frac{\Gamma^A, \phi_i^A : profile \qquad \Gamma^A, \phi_i^A \vdash \psi_j^B}{\Gamma^A, \phi_i^A < \psi_j^B : profile}\ \text{Dependency Insertion}$$

$$\frac{\Gamma^A : profile \qquad \vdash \psi_j^B}{\Gamma^A; \psi_j^B : profile}\ \text{Profile Extension}$$

Fig. 1. The System `(un)SecureND`: Profile Construction Rules

typing declaration $: profile$ to state that a formal expression can be considered a valid installation profile. By Empty Profile, an installation profile can be empty (base case); by Package Insertion, the elements in an installation profile are packages; by Dependency Insertion, a profile can be extended by satisfied dependencies; by Profile Extension, if a package can be validly executed in an empty profile, it can be added to an existing profile. Notice that unnecessary packages from any repository can still be added: this is possible for packages without dependencies through the Profile Extension rule, but more in general by an application of the Weakening Rule (see Figure 4). The result of such a profile extension is denoted by $\Gamma^A; \phi^B$ and $\Gamma^A; \Gamma^B$. It is worth noting that Weakening will preserve profile consistency as it requires additionally an instance of the *trust* rule (see Figure 3).

3.2 Rules for package execution

The operational rules in Figure 2 formulate compositionality of package execution. A judgement of the form $\Gamma^A \vdash \phi^B$ says that package ϕ from repository B is executable without errors within an installation profile with packages coming from repository A.

The rule *Atom* establishes valid package execution within the same installation profile and across repositories with satisfied dependencies. In the present version we assume $A < B$. $\bot$ says that if a profile is inconsistent, any package whatsoever can be executed. $\wedge$-I allows composition of packages from distinct profiles; by $\wedge$-E, each composing package can be obtained from the combined profiles (with $I = \{A, B\}$). $\vee$-I says that a combined profile can access any package from each of the composing profiles; by the elimination $\vee$-E, each package consistently inferred by each individual profile can also be executed under the extended profile. $\rightarrow$-Introduction expresses inference of a package from a combined profile as inference between packages (Deduction Theorem); its elimination $\rightarrow$-E allows to recover such inference as profile extension (Modus Ponens).

$$\frac{\Gamma^A;\Gamma^B : profile}{\Gamma^A;\Gamma^B \vdash \psi_i^B}\ \text{Atom, for any } \psi_i^B \in \Gamma^B \qquad \frac{\Gamma^A \vdash \bot}{\Gamma^A \vdash \phi^B}\ \bot$$

$$\frac{\Gamma^A \vdash \phi_i^A \qquad \Gamma^B \vdash \phi_j^B}{\Gamma^A;\Gamma^B \vdash \phi_i^A \wedge \phi_j^B}\ \wedge\text{-I} \qquad \frac{\Gamma^A;\Gamma^B \vdash \phi_i^A \wedge \phi_j^B}{\Gamma^A;\Gamma^B \vdash \phi_{i/j}^I}\ \wedge\text{-E}$$

$$\frac{\Gamma^A;\Gamma^B \vdash \phi_{i/j}^I}{\Gamma^A;\Gamma^B \vdash \phi_i^A \vee \phi_j^B}\ \vee\text{-I} \qquad \frac{\Gamma^A;\Gamma^B \vdash \phi_i^A \vee \phi_j^B \qquad \phi_{i/j}^{I\in\{A,B\}} \vdash \psi_k^C}{\Gamma^A;\Gamma^B \vdash \psi_k^C}\ \vee\text{-E}$$

$$\frac{\Gamma^A;\phi_i^B \vdash \phi_j^C}{\Gamma^A \vdash \phi_i^B \to \phi_j^C}\ \to\text{-I} \qquad \frac{\Gamma^A \vdash \phi_i^B \to \phi_j^C \qquad \Gamma^A \vdash \phi_i^B}{\Gamma^A;\phi_i^B \vdash \phi_j^C}\ \to\text{-E}$$

Fig. 2. The System `(un)SecureND`: Operational Rules

3.3 Access Rules

In Figure 3 we present the access rules. These allow a user's installation profile to act on packages available from a distinct repository. In particular, we formulate a rule to query a package from a repository (*read*) and one to install a package within a profile (*write*). A third rule is formulated to guarantee that only packages consistent with the installation profile can be installed (*trust*).

read says that from any consistent profile Γ^A a package ϕ_i^B can be read provided its dependencies are satisfied (if any). *trust* works as an elimination rule for *read*: it says that if a package ϕ_i^B can be read and it preserves profile consistency, then it can be trusted. *write* works as an elimination rule for *trust*: it says that a readable and trustable package can be installed. *exec* says that every package that is safely installed in a consistent profile can be executed in it. The Introduction rule for distrust DTrust-I expresses the principle that a package ϕ_i^B non-consistent with its installation profile can be negated to be trustworthy; the corresponding elimination DTrust-E uses $\to$-introduction to induce *write* of any package consistent with the conflict resolution. The Introduction rule for mistrust MTrust-I says that trust is removed for local packages conflicting with an intended installation (a queried package); the corresponding MTrust-E allows to trust any package which is consistent with the conflict resolution by removal of the mistrusted package in the installation profile. This holds for any required dependency in other repositories, as expressed by the side condition that requires checking for any $C < B$. By the latter set of rules, *distrust* is a flag for preventing installation of conflicting external packages, while *mistrust* is a flag for facilitating removal of conflicting packages present in the installation profile. Notice that both untrust functions are triggered by the querying operation on a repository, hence conflicts are highlighted before installation.

$$\frac{}{\Gamma^A \vdash Read(\phi_i^B)}\ read$$

$$\frac{\Gamma^A \vdash Read(\phi_i^B) \qquad \Gamma^A; \phi_i^B : profile}{\Gamma^A \vdash Trust(\phi_i^B)}\ trust$$

$$\frac{\Gamma^A \vdash Read(\phi_i^B) \qquad \Gamma^A \vdash Trust(\phi_i^B)}{\Gamma^A \vdash Write(\phi_i^B)}\ write \qquad \frac{\Gamma^A \vdash Write(\phi_i^B)}{\Gamma^A \vdash \phi_i^B}\ exec$$

$$\frac{\Gamma^A \vdash Read(\phi_i^B) \to \bot}{\Gamma^A \vdash \neg Trust(\phi_i^B)}\ \text{DTrust-I}$$

$$\frac{\Gamma^A \vdash \neg Trust(\phi_i^B) \qquad \Gamma^A \vdash \neg Trust(\phi_i^B) \to \psi_j^C}{\Gamma^A \vdash Write(\psi_j^C)}\ \text{DTrust-E}$$

$$\frac{\Gamma^A \vdash Read(\psi_i^B) \to \bot \qquad \Gamma^A \setminus \{\phi_j^A\} : profile}{\Gamma^A \setminus \{\phi_j^A\}; \psi_i^B \vdash \neg Trust(\phi_j^A)}\ \text{MTrust-I}$$

$$\frac{\Gamma^A \setminus \{\phi_j^A\}; \psi_i^B \vdash \neg Trust(\phi_j^A) \qquad \Gamma^C; \psi_i^B : profile}{\Gamma^A \setminus \{\phi_j^A\}; \Gamma^C \vdash Trust(\psi_i^B)}\ \text{MTrust-E, } \forall C < B$$

Fig. 3. The System `(un)SecureND`: Access Rules

3.4 Structural Rules

Structural rules hold with restrictions for (`un`)`SecureND`, see Figure 4. As a result the system qualifies as substructural, see e.g. [14].

Weakening is constrained by an instance of *trust*: it says that a valid installation of ϕ_i^A is preserved under a profile extension in view of a trusted package ϕ_j^B, i.e. one whose profile extension is provably consistent.

Contraction is constrained by preservation of package ordering: it says that a valid installation of ϕ_k^A is preserved when removing an instance of identical packages $\phi_i^A; \phi_i^B$, provided one preserves the package from the higher repository in the order dependency, so as to guarantee any further dependency below.

Exchange is doubly constrained by order: it says that a valid installation of ϕ_k^A is preserved under reorder of packages ϕ_i, ϕ_j, if those come from the same repository A and if there is no involved dependency between them.

Finally, the Cut rule expresses valid package execution under profile extension: if a package ϕ_i^B is validly executed under profile Γ^A and a profile Γ^B including ϕ_i^B allows execution of a package ϕ_j^B, then the extended profile $\Gamma^A; \Gamma^B$ allows execution of ϕ_j^B.

$$\frac{\Gamma^A \vdash Write(\phi_i^A) \qquad \Gamma^A \vdash Trust(\phi_j^B)}{\Gamma^A; \phi_j^B \vdash Write(\phi_i^A)}\ \text{Profile Weakening}$$

$$\frac{\Gamma^A, \phi_i^A; \phi_i^B \vdash Write(\psi_k^A) \qquad A < B}{\Gamma^A, \phi_i^A \vdash Write(\psi_k^A)}\ \text{Profile Contraction}$$

$$\frac{\Gamma^A, \phi_i^A, \phi_j^A \vdash Write(\phi_k^A) \qquad \phi_i^A \not< \phi_j^A}{\Gamma^A, \phi_j^A, \phi_i^A \vdash Write(\phi_k^A)}\ \text{Profile Exchange}$$

$$\frac{\Gamma^A \vdash \phi_i^B \qquad \Gamma^B, \phi_i^B \vdash \phi_j^B}{\Gamma^A; \Gamma^B \vdash \phi_j^B}\ \text{Profile Cut}$$

Fig. 4. The System `(un)SecureND`: Structural Rules

4 The Distrusted Uninstall Problem

Consider a profile $\Gamma^A = \{\phi_1^A < \ldots < \phi_n^A\}$ and a package ϕ_m^B which one wishes *not to install*. This might be due to a security constraint, or an explicit conflict in view of an installed package $\phi_i^A \in \Gamma$, which one explicitly wants to preserve. We call such a package ϕ_m^B *distrusted*. In the calculus, this corresponds to the conclusion of the DTrust-I rule

$$\Gamma^A \vdash \neg Trust(\phi_m^B)$$

The *Distrusted Uninstall Problem* is to determine which packages can be installed in Γ^A that do not depend on ϕ_m^B. Our formulation allows to express this principle as the request to obtain the maximal set of formulas $\{\psi_i^N\}$ from any repository $N \geq B$ such that

$$\Gamma^A \vdash \neg Trust(\phi_m^B) \rightarrow \{\psi_i^N\}$$

By DTrust-E, this guarantees the right to install ψ_i^N. The first step consists in transforming our problem in a formulation that removes the trust condition.

Lemma 1. $\Gamma^A \vdash \neg Trust(\phi_m^B) \rightarrow \psi_i^N$ *iff* $\Gamma^A; \neg\phi_m^B \vdash \psi_i^N$.

Proof. For the left-to-right direction: By the assumption $\Gamma^A \vdash \neg Trust(\phi_m^B)$ and consistency of negation, $\Gamma^A \vdash Trust(\neg\phi_m^B)$; similarly, from the premise $\Gamma^A \vdash \neg Trust(\phi_m^B) \rightarrow \psi_i^N$ and consistency of negation we get $\Gamma^A \vdash Trust(\neg\phi_m^B) \rightarrow \psi_i^N$. Now apply *write* to $Trust(\neg\phi_m^B)$ and eliminate the function through *exec*; by $\rightarrow$-E we obtain $\Gamma^A; \neg\phi_m^B \vdash \psi_i^N$.

For the right-to-left direction: By the assumption $\Gamma^A; \neg\phi_m^B \vdash \psi_i^N$ it holds $\Gamma^A; \neg\phi_m^B : profile$, which justifies $\Gamma^A \vdash Read(\neg\phi_m^B)$ by *read*, $\Gamma^A \vdash Trust(\neg\phi_m^B)$ by the previous and *trust* and $\Gamma^A \vdash \neg Trust(\phi_m^B)$ by $\neg$-distribution. It follows $\Gamma^A; \neg Trust(\phi_m^B) \vdash \psi_i^N$ by substitution from the assumption, and $\Gamma^A \vdash \neg Trust(\phi_m^B) \rightarrow \psi_i^N$ is obtained by $\rightarrow$-I.

We can now reduce the latter to an operation on all packages coming from the repository involved by the distrust operation:

Lemma 2. *If* $\Gamma^A; \neg\phi_m^B \vdash \psi_i^N$ *then* $\Gamma^A; \Gamma^B \setminus \{\phi_m^B\} \vdash \psi_i^N$*, for all consistent profiles* Γ^B *that include* ϕ_m^B*.*

Proof. Γ^A can be extended with every consistent package from B; by definition $\Gamma^A; \neg\phi_m^B \vdash \neg Trust(\phi_m^B)$, hence by Weakening this is possible except for ϕ_m^B as it does not satisfy *trust*.

The above corresponds to finding the maximal set of formulas in Γ^B that allows to execute ψ_i^N without requiring ϕ_m^B in the profile. To this aim, it is enough to find all $\phi_l^B \not> \phi_m^B$, i.e. the set of packages in B that have no dependencies from ϕ_m^B.

What has been so far restricted to one repository, can now be generalised to any repository that preserves the dependency condition:

Lemma 3. $\Gamma^A; \phi_l^N \vdash Write(\psi_i^N)$ *iff* $(\phi_l^N \not< \xi_m^N \not< \psi_i^N)$ *for any distrusted package* ξ_m^N *and any repository* $N > A$*.*

Proof. For the right-to-left direction. Assume the following: $\Gamma^A; \phi_l^N \vdash Write(\phi_i^N)$ and $\Gamma^A; \phi_l^N \vdash \neg Trust(\phi_m^N)$. Then: if $\phi_l^N < \phi_m^N$, then $\Gamma^A; \phi_l^N \vdash \phi_m^N$ by Atom, contradicting the distrust assumption; and if $\phi_m^N < \phi_i^N$ then similarly $\phi_m^N \vdash \phi_i^N$ and by Weakening it is possible to obtain $\Gamma^A; \phi_l^N, \phi_m^N \vdash Write(\phi_i^N)$, again contradicting the distrust assumption.

For the left-to-right direction. Assume $(\phi_l^N \not< \phi_m^N \not< \phi_i^N)$ and $\Gamma^A; \phi_l^N \vdash \neg Trust(\phi_m^N)$. Then: because $\phi_l^N \not< \phi_m^N$, the second assumption above does not require to remove ϕ_l^N as by Lemma 2; and because $\phi_m^N \not< \phi_i^N$, installing the latter does not require installing the former. Hence $\Gamma^A; \phi_l^N \vdash Write(\phi_i^N)$ holds.

Finally, our main result is obtained:

Theorem 1. ***(Distrusted Uninstall)*** *Given a package* ϕ_m^B *distrusted under profile* Γ^A*, a package* ψ_i^N *can be installed in* Γ^A *iff* $\phi_m^B \not< \psi_i^N$*.*

Proof. From Definition 2 and Lemma 3 by substitution.

This last result identifies distrusted packages as those that have at least a dependency from one package conflicting with the current installation profile.

5 The Mistrusted Uninstall Problem

Consider a profile $\Gamma^A = \{\phi_1^A < \ldots < \phi_n^A\}$ and a package ϕ_m^B which one wishes to install in it: in the calculus, this corresponds to the conclusion of an instance of the Write rule, $\Gamma^A \vdash Write(\phi_m^B)$. Assume that ϕ_m^B is in conflict with the given profile

$$\Gamma^A \vdash Read(\phi_m^B) \to \bot$$

The *Mistrusted Uninstall Problem* is to determine the set $\Phi^A = \{\phi_i^A \in \Gamma^A \mid \phi_i^A \rightarrow \neg\phi_m^B\}$ which should be removed when installing ϕ_m^B. We will call any such package ϕ_i^A a *mistrusted package*. Hence the problem is to identify the minimal set of formulas Φ^A such that for each $\phi_i^A \in \Phi^A$

$$\Gamma^A \setminus \Phi^A; \phi_m^B \vdash \neg Trust(\phi_i^A)$$

and by MTrust-E, given any other set of formulas Γ^C required by ϕ_m^B, it allows

$$\Gamma^A \setminus \Phi^A; \Gamma^C \vdash Trust(\phi_m^B)$$

We start by identifying the minimal subset of packages from the current installation profile that satisfies the conflict:

Lemma 4. *If $\Gamma^A \vdash Read(\phi_m^B) \rightarrow \bot$, then $\exists \Phi^A \subseteq \Gamma^A$ such that $\Phi^A = \{\phi_i^A < \ldots < \phi_n^A\} \vdash Read(\phi_m^B) \rightarrow \bot$.*

Proof. $\forall \phi_i^A, \phi_j^A \in \Gamma^A$, if $\phi_i^A \vdash Read(\phi_m^B) \rightarrow \bot$ and $\phi_i^A < \phi_j^A$, then $\phi_j^A \vdash Read(\phi_m^B) \rightarrow \bot$. And $\forall \phi_h^A < \phi_i^A$, $\phi_h^A \vdash Read(\phi_m^B)$. Hence it suffices to identify the maximal ϕ_i^A in conflict with ϕ_m^B and to include it in Φ^A together with all packages in Γ^A that depend on it. We will call Φ^A a *maximally mistrusted set*.

Lemma 5. *Consider a maximally mistrusted $\Phi^A \subseteq \Gamma^A$ such that $\Phi^A \vdash Read(\phi_m^B) \rightarrow \bot$ as of Lemma 4. Then $\forall \phi_i^A \in \Phi^A$, $\phi_i^A < Read(\phi_m^B) \rightarrow \bot$.*

Proof. This holds by construction of Φ^A in Lemma 4 and the Dependency Insertion Rule.

Lemma 6. *If $\phi_i^A \vdash Read(\phi_m^B) \rightarrow \bot$, then $\phi_i^A \not< \phi_m^B$.*

Proof. Starting from $\phi_i^A \vdash Read(\phi_m^B) \rightarrow \bot$ we apply D-Trust-I, $\neg$-distribution, *write* and *exec* to obtain $\phi_i^A \vdash \neg\phi_m^B$, from which we obtain $\phi_i^A < \neg\phi_m^B$ from Dependency Insertion and $\phi_i^A \not< \phi_m^B$ by contrapposition.

Theorem 2. ***(Mistrusted Uninstall)*** *Given a package ϕ_m^B to be installed under profile Γ^A, a package ϕ_i^A is mistrusted in Γ^A iff for all $\Gamma^A \subseteq \{\phi_i^A < \phi_j^A\}$*

1. *$\Gamma^A \vdash \phi_j^A \rightarrow \neg\phi_m^B$,*
2. *$\phi_j^A < Read(\phi_m^B) \rightarrow \bot$ and*
3. *$\phi_i^A \not< \phi_m^B$.*

Proof. The first condition is required by Lemma 5 to include all the dependencies in the maximally mistrusted set. The second condition holds from Lemma 6. Finally, the third condition holds by contradiction: if $\phi_i^A < \phi_m^B$, then $\phi_i^A \vdash \phi_m^B$ by Dependency Insertion; it follows by Weakening that $\phi_i^A; \phi_m^B : profile$ and hence $\phi^B \vdash Trust(\phi_i^A)$.

This last result identifies packages to be removed as those that are in maximally mistrusted set and do not satisfy any dependency for the package to be installed under the current profile.

6 An Example

Consider the simple scenario presented in Section 1 where a user has the following installation profile:

$$\Gamma^{m-f-nf} \left\{ \begin{array}{l} \Gamma^{main} = \{\phi_1^m, \phi_2^m\} \\ \Gamma^{free} = \{\psi_1^f\} \\ \Gamma^{nonfree} = \{\xi_1^{nf}\} \end{array} \right\}$$

with the following dependencies

$$\Gamma^{m-f-nf} \left\{ \begin{array}{l} \phi_1^m < \psi_1^f \\ \phi_2^m < \psi_1^f \\ \psi_1^f < \xi_1^{nf} \end{array} \right\}$$

Assume the user distrusts a package ψ_n^f, e.g. because it is considered harmful or unsecure. The Distrusted Uninstall Problem asks which packages can be further installed in Γ^{m-f-nf} without installing ψ_n^f. Consider now a package $\psi_2^f \not> \psi_n^f$, then the following derivation holds:

$$\frac{\dfrac{\text{D}}{\Gamma^{m-f-nf} \vdash \neg Trust(\psi_n^f)} \qquad \dfrac{\text{D'}}{\Gamma^{m-f-nf} \vdash Read(\psi_2^f)} \qquad \psi_n^f \not< \psi_2^f}{\Gamma^{m-f-nf} \vdash Write(\psi_2^f)}$$

In other words, flagging ψ_n^f as distrustful does not impede the installation of a package ψ_2^f if the latter does not depend on the former.

Assume moreover that the user wishes to install an additional package $\xi_2^{nf} > \phi_1^m$, but such that $\phi_2^m \vdash Read(\xi_2^{nf}) \rightarrow \bot$: in other words, ξ_2^{nf} depends on ϕ_1^m, but is in conflict with ϕ_2^m (which is possible, given the latter does not depend on ϕ_1^m). Then assuming a package ψ_2^f replacing the functionalities of ϕ_2^m, the following derivation holds:

$$\frac{\dfrac{\phi_2^m \vdash Read(\xi_2^{nf}) \rightarrow \bot \qquad \phi_2^m < \psi_1^f}{\Gamma^{m-f-nf} \setminus \{\phi_2^m < \psi_1^f\}; \xi_2^{nf} \vdash \neg Trust(\phi_2^m < \psi_1^f)} \qquad \psi_2^f; \xi_2^{nf} : profile}{\Gamma^{m-f-nf} \setminus \{\phi_2^m < \psi_1^f\}; \xi_2^{nf} \vdash Write(\psi_2^f)}$$

In other words the installation of ξ_2^{nf} requires removing $\phi_2^m < \psi_1^f$ and it is compatible with the installation of ψ_2^f.

7 Conclusions

In this paper we have formulated two variants to the Uninstall Problem. Each relies on a different semantic qualification of untrusted packages required to be removed or prevented from installation in a given installation profile, in order to preserve consistency.

Our approach is grounded on the logic `(un)SecureND`, including an explicit *trust* function on formulas to guarantee consistency check at each retrieval step (after a *read* function), before installation rights are granted for a package (by a *write* function). The fragment of the language presented in this paper allows to express negation over trust as a dis-installation requirement. Different pairs of introduction/elimination rules determine the selection of one of two resolution strategies: one flags a package external to the installation profile as distrusted and hence as not installable; the other identifies already installed packages to be removed. The selection takes care of identifying and removing all required dependencies. We have illustrated the working protocol through an easy example. As already mentioned, validation of the system is obtained by implementation of the `(un)SecureND` calculus as a large inductive type in the Coq proof assistant. The development is available at `https://github.com/gprimiero/SecureNDC`.

A characteristic of the logic `(un)SecureND` is its substructural nature, which in future work can be exploited to investigate cases of strengthened and limited resource redundancy for fault tolerance and source shuffling for security. Other applications of negative trust can be investigated to distinguish between malevolent and simply unsuccessful sources.

References

1. Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Dependency solving: A separate concern in component evolution management. *Journal of Systems and Software*, 85(10):2228–2240, 2012.
2. Pietro Abate, Roberto DiCosmo, Ralf Treinen, and Stefano Zacchiroli. MPM: A Modular Package Manager. In *Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering*, CBSE '11, pages 179–188, New York, NY, USA, 2011. ACM.
3. Daniel Le Berre and Anne Parrain. On SAT Technologies for Dependency Management and Beyond. In Steffen Thiel and Klaus Pohl, editors, *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops)*, pages 197–200. Lero Int. Science Centre, University of Limerick, Ireland, 2008.
4. Jaap Boender. Formal verification of a theory of packages. *ECEASST*, 48, 2011.
5. Jaap Boender, Giuseppe Primiero, and Franco Raimondi. Minimizing transitive trust threats in software management systems. In Ali A. Ghorbani, Vicenç Torra, Hüseyin Hisil, Ali Miri, Ahmet Koltuksuz, Jie Zhang, Murat Sensoy, Joaquín García-Alfaro, and Ibrahim Zincir, editors, *13th Annual Conference on Privacy, Security and Trust, PST 2015, Izmir, Turkey, July 21-23, 2015*, pages 191–198. IEEE, 2015.

6. Ramanathan V. Guha, Ravi Kumar, Prabhakar Raghavan, and Andrew Tomkins. Propagation of trust and distrust. In *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, pages 403–412, 2004.
7. Alexey Ignatiev, Mikoláš Janota, and Joao Marques-Silva. Towards Efficient Optimization in Package Management Systems. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 745–755, New York, NY, USA, 2014. ACM.
8. Audun Jøsang and Simon Pope. Semantic Constraints for Trust Transitivity. In Sven Hartmann and Markus Stumptner, editors, *APCCM*, volume 43 of *CRPIT*, pages 59–68. Australian Computer Society, 2005.
9. Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the Complexity of Large Free and Open Source Package-Based Software Distributions. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*, pages 199–208. IEEE Computer Society, 2006.
10. Stephen Marsh and MarkR. Dibben. Trust, Untrust, Distrust and Mistrust – An Exploration of the Dark(er) Side. In Peter Herrmann, Valérie Issarny, and Simon Shiu, editors, *Trust Management*, volume 3477 of *Lecture Notes in Computer Science*, pages 17–33. Springer Berlin Heidelberg, 2005.
11. D. Harrison McKnight and Norman L. Chervany. Trust and Distrust Definitions: One Bite at a Time. In Rino Falcone, Munindar P. Singh, and Yao-Hua Tan, editors, *Trust in Cyber-societies, Integrating the Human and Artificial Perspectives*, volume 2246 of *Lecture Notes in Computer Science*, pages 27–54. Springer, 2000.
12. Giuseppe Primiero. A Calculus for Distrust and Mistrust. In Sheikh Mahbub Habib, Julita Vassileva, Sjouke Mauw, and Max Mühlhäuser, editors, *Trust Management X - 10th IFIP WG 11.11 International Conference, IFIPTM 2016, Darmstadt, Germany, July 18-22, 2016, Proceedings*, volume 473 of *IFIP Advances in Information and Communication Technology*, pages 183–190. Springer, 2016.
13. Giuseppe Primiero and Franco Raimondi. A typed natural deduction calculus to reason about secure trust. In Ali Miri, Urs Hengartner, Nen-Fu Huang, Audun Jøsang, and Joaquín García-Alfaro, editors, *2014 Twelfth Annual International Conference on Privacy, Security and Trust, Toronto, ON, Canada, July 23-24, 2014*, pages 379–382. IEEE, 2014.
14. Greg Restall. *An Introduction to Substructural Logics*. Routledge, 2000.
15. C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. OPIUM: Optimal Package Install/Uninstall Manager. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 178–188, 2007.
16. Jérôme Vouillon and Roberto Di Cosmo. On Software Component Co-installability. *ACM Trans. Softw. Eng. Methodol.*, 22(4):34:1–34:35, October 2013.
17. Cai-Nicolas Ziegler and Georg Lausen. Propagation Models for Trust and Distrust in Social Networks. *Information Systems Frontiers*, 7(4-5):337–358, December 2005.