Negative trust for conflict resolution in software management

Giuseppe Primiero
Department of Philosophy, University of Milan

Jaap Boender
Department of Computer Science, Middlesex University London

November 7, 2018

Abstract

Software management systems need to preserve integrity by the handling, approval, tracking and execution of changes on the packages of the current installation profile. This is a problematic task, which needs to be accounted for both in terms of installation of new packages and removal of conflicting ones. While existing approaches are able to identify dependency satisfaction and conflicts, a broader and efficient approach can be formalised in terms of trust. Positive instances of trust are required by the identification of safely installable packages. Negative trust, a much less explored concept, can be useful to analyse the complementary issue of packages' removal both in case of conflicts and of security issues. In this paper we develop a logic of negative trust with two aims: identifying packages that are undesirable in view of the current installation profile; and currently installed packages that become inconsistent with a new intended installation. The logic provides distinct procedures for the identification of either case. We illustrate properties of the calculus, provide a simple working example and offer a translation of the protocol to the Coq proof assistant for verification of its formal correctness.

1 Introduction

Software management configuration is among the most pervasive problems in modern personal computing. The maintenance of a consistent and functioning software system is complicated by the ability of modern systems to accommodate multiple users, by the need of releases to preserve package compatibility across versions and customizations, and the essential coherence required by distributed systems, especially in the context of web and cloud services. One of the specific aspects involved by these issues is software change management: the handling, approval, tracking and execution of changes on the packages of the current installation profile, with the crucial requirement that integrity of the system

be preserved. In most cases, users manage their installation profile through a software package system which satisfies integrity through an underlying logical notion of validity.

Definition 1 (Valid Installation profile)**.** *An installation profile is valid if and only if all dependencies for the installed packages are met and all conflicts are avoided.*

The aim of software management is to preserve validity, which can be affected by installation of new packages. Let us illustrate this issue with a simple example. The server-side software underlying the package management system in the free operating system Debian and its numerous derivatives is called **dpkg**; the user typically accesses it through the **apt** tool. Binary packages are capable of declaring their dependencies and conflicts and this allows the system to maintain profile validity.[1] When a conflict is declared between two packages, **dpkg** forbids them to be unpacked on the system at the same time. So a package ϕ conflicts with package ψ when ϕ will not operate if ψ is installed on the system. If one of the two is already installed in the profile, then it must be removed before the other one can be unpacked. If the package intended for installation is marked as replacing another one on the system, or the one on the system is marked as deselected, then **dpkg** will remove the package which is causing the conflict. If both packages are marked Essential, it will halt the installation of the new package with an error. This makes sure that essential packages are not substituted by non-essential ones. This also illustrates that packages and their dependencies come with a priority relation. Note also that sharing functionalities with another package is not a sufficient reason to declare conflicts with that package.

1.1 An example

Let us consider the dependencies list for the Tor package from Figure 1: this list presents an underlying conflict with `libsl0.9.8` which, as shown in Figure 2, is an outdated package version which is no longer available in the repository: this conflict could be resolved by manual installation or just by admitting the required higher version `libsl1.0.0` of the library involved by the conflict. Moreover, the Tor Bundle depends on the package `libc6`, which is the C Gnu library required by any program in that language.

Consider now that a user under this same installation profile attempts an installation of the package triggering the dependencies listed in Figure 3: this package induces the upgrade and associated error with `libc6` illustrated in Figure 4.[2] The software package `hedgewars` requires an upgrade on `libc6`; this would require overwriting it with its replacement; but the library cannot be (temporarily) removed, as it is required by `Tor`. In other words, `libc6` has unresolvable dependencies while, at the same time, removing it becomes

[1] `https://www.debian.org/doc/debian-policy/ch-relationships.html\#s-conflicts` for details.

[2] This error is reported at `https://ubuntuforums.org/showthread.php?t=1091866`.

```
gprimiero@xps:~ apt-cache depends tor
tor
  Depends: libc6
  Depends: libevent-2.0-5
  Depends: libseccomp2
  Depends: libssl1.0.0
  Depends: libsystemd0
  Depends: zlib1g
  Depends: adduser
  Depends: init-system-helpers
  Depends: lsb-base
  Conflicts: <libssl0.9.8>
  Recommends: logrotate
  Recommends: tor-geoipdb
  Recommends: torsocks
  Suggests: mixmaster
  Suggests: torbrowser-launcher
  Suggests: socat
  Suggests: tor-arm
  Suggests: apparmor-utils
  Suggests: obfsproxy
  Suggests: obfs4proxy
```

Figure 1: List of dependencies for Tor Package.

```
gprimiero@xps:~ sudo apt-get install libssl0.9.8
  Error:

  Reading package lists... Done
  Building dependency tree
  Reading state information... Done
  Package libssl0.9.8 is not available, but is referred to by another package.
  This may mean that the package is missing, has been obsoleted, or
  is only available from another source

  E: Package 'libssl0.9.8' has no installation candidate
```

Figure 2: Conflict message for obsolete package libssl0.9.8.

hard due to the number of other packages that depend on it, including the Tor Bundle.

This example illustrates that an optimal conflict resolution strategy would make the user not only aware that `libssl0.9.8` is not safe to be installed in view of the direct conflict with `Tor` (as indeed **apt** does); but also that `hedgewars` is non-safe in view of the current installation, because installing it would require `Tor` to be removed, due to its dependency in view of the given version of `libc6`.

Determining these consistency relations between packages in a given installation profile is essential for the system's stability, but also to prevent the possibility of security threats in critical systems: if the system at hand is meant to control infrastructure rather than allow to play games, unsafety means a lot more.

1.2 The Uninstall Problems

The ability to determine which and how many software packages would need to be removed by the installation of a new one, is therefore an essential aspect

```
gprimiero@xps:~ sudo apt-get install hedgewars

Upgraded the following packages:
fp-compiler (2.2.0-dfsg1-9ubuntu1) to 2.2.2-8
fp-units-base (2.2.0-dfsg1-9ubuntu1) to 2.2.2-8
fp-units-misc (2.2.0-dfsg1-9ubuntu1) to 2.2.2-8
fp-units-rtl (2.2.0-dfsg1-9ubuntu1) to 2.2.2-8
libc6 (2.8~20080505-0ubuntu9) to 2.9-4
libc6-amd64 (2.8~20080505-0ubuntu9) to 2.9-4
libc6-dev (2.8~20080505-0ubuntu9) to 2.9-4
libc6-dev-amd64 (2.8~20080505-0ubuntu9) to 2.9-4
libc6-i686 (2.8~20080505-0ubuntu9) to 2.9-4
```

Figure 3: Requirement for installation of hedgewars package

```
gprimiero@xps:~

Unpacking replacement libc6 ...
dpkg: error processing /var/cache/apt/archives/libc6_2.9-4_i386.deb
 (--unpack):
 trying to overwrite '/usr/share/man/man1/localedef.1.gz',
 which is also in package belocs-locales-bin
dpkg-deb: subprocess paste killed by signal (Broken pipe)
Processing triggers for man-db ...
Errors were encountered while processing:
 /var/cache/apt/archives/libc6_2.9-4_i386.deb
```

Figure 4: Conflict message for required libc6 library.

of software management. This not only to provide confidence on the safety of the system, but also to allow the user to asses the risks and changes required. The problem of maintaining profile consistency and system integrity in view of uninstall processes has already been addressed in the literature. In [50], the problem is presented in the following terms:

Definition 2. *(**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 $\psi_1, \ldots, \psi_n$ currently installed that are in conflict with the intended installation of ϕ and all of its dependencies $\xi_1, \ldots, \xi_n$. This version of the problem can be complemented by that of identifying packages that depend on an undesired one.

Definition 3. *(**Uninstall Problem (Alternative Version)**). Given a new package ϕ whose installation needs to be prevented, determine for which other packages depending on ϕ installation has to be prevented.*

This means identifying all packages $\xi_1, \ldots, \xi_n$ depending on the package ϕ that is in conflict with some of the currently installed packages $\psi_1, \ldots, \psi_n$ and for which installation needs to be prevented.

The relations between client and software repository illustrated by these problems can be reformulated in terms of *trust*. In particular, in the example from the previous subsection, we are considering instances of *negative trust*: if trust is granted to a package which preserve profile validity, then trust should be denied to any package that threatens such consistency (and completeness).

Whatever the nature of the interested property breach, we can express this functionally as a trust denial operation. In our example, `libssl0.9.8` is strictly untrustworthy under the current installation; and, provided the intended installation of `hedgewars`, the package `Tor` looses its trustworthiness status, provided it depends on `libc6` which would need to be removed. Obviously, this relation is instantiated also in a different form: `hedgewars` should be labelled untrustworthy, and more so than `libssl0.9.8`, as the damage it induces to system's stability is so much more vast.

This illustrates how an account of the effects of conflicts is essential, as well as a description of how procedurally different operations can be performed to resolve the same conflict. Negative instances of trust have a complex, and so far little studied, meaning. One way to clarify it is by clearly distinguishing between two of its forms: here and in the following the term *untrust* is used as neutral for 'negative trust' with respect to its derivatives *mistrust* and *distrust*, where the former expresses trust removal, the latter trust denial. These two types of operations can be used to formalize different approaches to conflict prevention and conflict resolution. On this basis, we adapt here the Uninstall Problems from Definitions 2 and 3 to the two semantics of untrust:

Definition 4. *(**Mistrusted Uninstall Problem**) Given a package ϕ to be installed but conflicting with the current profile, determine which packages have to be mistrusted in order for ϕ to become installable.*

As in the approach from [50], we are interested here in determining the minimal set of packages inconsistent with ϕ that eventually have to be removed from the installation profile.

Definition 5. *(**Distrusted Uninstall Problem**) Given a distrusted package ϕ, determine which other packages can be installed (i.e. do not depend on ϕ).*

In this case, we are obviously interested in determining the maximal set of installable packages that do not conflict with ϕ and therefore eventually can be granted installation.

In the present paper we provide a solution to these two problems in software management through their formalization in a logic for *negative trust*. Consistency-checking on contents is a natural basic way to interpret computational trust: if this is translated in the context of software management, trust corresponds to a property obtained by installation validity as per Definition 1. Negative trust seems therefore an appropriate functional counterpart to be applied to any software package which breaks such validity. Note that this can notion of negative trust must allow to express the two cases mentioned above, of trust removal (from the current profile) ad of trust denial (for external packages).

To express such operations, we are in need of a language which expresses trust and its negation as functions on contents (i.e. software packages in the present context), rather than as a relation between agents. To this aim the logic `(un)SecureND`, introduced in Section 4, is an optimal tool. The logic allows to

reason about statements expressing the following situation: a package ϕ can be consistently installed under profile Γ; while installation for a given conflicting package ψ is prevented or ψ is accommodated by removal of existing packages in Γ. Positive trust applies to software packages which are safe to install and conflicts are prevented through negative trust. The kind of inconsistencies that can be treated through this formal machinery are not just those induced by technical requirements of the packages, but can be extended also to security and ownership issues. This formal strategy has the conceptual advantage to split the uninstall problem presented in Definition 2 in the two easier, clearer versions, where the uninstall process is qualified with respect to whether the removal operation has to happen on the client or on the server side. Moreover, our approach offers also a computable approach to trust management to reduce risks related to installation profile inconsistency.

This approach is novel from a conceptual point of view, because software dependency satisfaction as trust management has not yet been 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. More specifically, a SAT solver can only give a response to the question whether a package is installable or not, i.e. whether its dependencies are satisfied and it does not conflict with any other packages that are already installed. Our approach is also able to deal with packages that are technically installable, but should nonetheless not be installed, for example because they have security issues or come from untrusted sources. Moreover, the procedural approach provided by the proof-theoretic semantics explicitly formulate the different strategies of conflict resolution corresponding to trust denial and trust removal. Ultimately, the ideal approach would be to combine SAT solving with our calculus, so that the resolution of dependencies and conflicts can be taken care of by a SAT solver, and issues of trust and provenance by a tool based on our approach.

The paper is an extension of the work presented in [42], including a detailed example, a more extensive analysis of the literature in the several related areas, an informal presentation of the protocols for negative trust and more extensive clarification of their formal counterpart in the logic `(un)SecureND`, and a more detailed presentation of the (now extended) Coq library. The paper is structured as follows. In Section 2 we offer an overview of related works in the areas of computational trust, software management and automated theorem proving for those two specific research fields. In Section 4 we introduce the system `(un)SecureND`, which provides the formal machinery for our analysis. In Section 5 the Mistrusted Uninstall Problem is reformulated within our logic and its solution illustrated. In Section 6 the same is done for the Distrusted Uninstall Problem. In Section 7 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. These are two extensively developed but largely non-overlapping research areas. The present work offers, to our knowledge, a first approach to software dependency management that makes use of a computational account of trust. Moreover, we do so through a formal proof-theoretic approach which also allows us to exploit the possibility of automatic proof checking for the purpose of proving correctness of the proposed protocol. In this section we briefly overview related works in both areas and compare those to our approach and results.

2.1 Software Management

The resolution of installation problems is essential for quality assurance of package based software distributions and their engineering. The present approach is located within the literature concerning globally managed, hierarchically organised software packages. The problem has been investigated largely in view of issues related to dependencies, interactions between software from different packages and upgrade [21]. SAT solving has been so far the most promising approach for the development of efficient methods of dependency graph resolution. SAT technology has been used in [37] to validate dependencies and check installability of packages of specific Linux distributions. In particular, this encoding has allowed to establish its complexity as NP-hard and it represents the basis for several quality metrics including the above mentioned issue of relevance in a repository, see also [1], and the strength of conflicts, see [20]. The analysis of conflicts and defective reporting are thus crucial, so is their analysis and possible resolution, see [8].

The *Uninstall Problem* from [50] is at the basis of the Opium package-management tool, also using 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.[3] A review of the state-of-the-art package managers and their ability to keep up with evolution and their dependency solving abilities is offered in [3], 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. The solvability of the decision problem related to software dependency management and its optimization are also considered in [9].

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, [33]. In [51] a

[3] 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/`.

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 [10].

This brief – and only partially complete – overview illustrates the variety and complexity of issues arising in the context of software dependency management. Our approach is a first attempt in a new direction in terms of trust management, and as such maintained at the lowest possible level of complexity. 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. This helps in that package removal from a profile always proceeds by identifying 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 indirectly with upgrades by using the more complex tactic of removing older versions first and installing newer ones afterwards. The issue of reciprocal dependencies, i.e. where two packages depend one from the other, is abstracted in the present formulation. The *Mistrusted Uninstall Problem* introduced in Section 1 and formally illustrated in Section 5 replicates the intuition of the co-installability problem in the setting where external packages (and their dependencies) are in explicit conflict with currently installed ones (and those they depend on).

2.2 Computational Trust

Computational Trust is an area of extensive work, with both foundational work and applications in security and reputation models, uncertain environments, autonomous systems and social networks. The major problems of interest are related to trust propagation, trust interference and distrust blocking, see e.g. [26, 12, 28, 54, 38, 35, 18, 36, 15, 25]. Applications can be found in Internet-based services [27], to component-based systems [29, 52], accuracy [7], trust transferability in context-aware [48] and mobile applications [49], and epistemic reliability [53]. Propagation for trust as a first order relation is largely studied [17, 14, 34, 6, 40], with solutions to undesirable effects ranging from decentralised trust [5], bounded-transitivity in authorization contexts [16], to guarantors-based constraints [18]. The problem of defining models of computational trust is also related to social attributes and values, also in connection to delegation decision, see e.g. [13, 30].

Our contribution is based on previous work both for a general logic of trust and for its application to software management problems. The proof-theoretic language `SecureND` was developed first in [43] to reason about secure instances of trust relations in the context of access control systems. The system has been later extended to the language `(un)SecureND` in [41] for instances of negative trust. The system has been applied then to the problem of trust transitivity for software management systems in the form of threats generated by transitive

authorizations over (possibly unreliable) software repository, see [11, 42]. Moreover, the generality of the definition of trust provided by the logic has allowed applications to information transmission in networks [44], to modelling of trust and reputation protocols for transmissions across ad hoc networks [45].

Negative trust notions are admittedly not largely present in the literature, with some exceptions. Approaches in the social sciences have typically accounted distrust as a response to lack of information [23, 24] and mistrust as former trust destroyed or healed [47]. In more recent accounts focusing on computational trust [38, 4], the straight first-order relation account of trust is qualified in terms of contextual situations. This weakening of the absolute trust relation induces the following definitions (we skip here formal details for brevity):

- mistrust is misplaced trust, i.e. a situation where there was a positive estimation and trust has been misplaced (not necessarily betrayed);
- untrust is little trust;
- distrust is no trust.

This approach designs a continuum between the positive and negative evaluations (with some blurry limit at trust value zero) but it abstracts from the reasons behind the attribution of these evaluations, in favour of a purely quantitative approach. In [31], a scale is considered

> *unjustified trust* (antitrust) → *justified trust* (sceptical) → *conditional trust* (contingent) → *unconditional trust* (faith)

which clearly relies strongly on evaluation criteria which are not only quantitative. Propagation for negative (first-order) trust is formulated in [35].

Our contribution relies on a strict distinction between *distrust* and *mistrust*, which we explain in some details below in Section 3.

2.3 Automated Theorem Proving

We also hope to facilitate the introduction of automated theorem provers in both areas of computational trust and software management. In particular, theorem provers can be beneficial in the process of checking intended installations in order to anticipate possible conflicts. Our work in [11] and the work presented in [3] present formal translation to libraries for the Coq theorem prover,[4] with the aim of verifying their 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.

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

3 Forms of (un)trust

The semantics of negated trust is here detailed in two versions, distinguishing between

- *distrust*: trust denied to software packages inducing a conflict with a currently installed package;
- *mistrust*: trust revoked to currently installed packages, in view of desired new packages to be installed.

Our approach formalises these cases for install and uninstall operations which, as far as we are aware, is entirely missing in the literature in computational trust. In large part of this tradition, (un)trustworthiness is a global property of agents, and (un)trust holds as a first-order relation between agents. This widely accepted view of (un)trust has various shortcomings, in particular it induces either acceptance or rejection of all activities from the agents designated as trustworthy or untrustworthy, and it generates unintentionally transitive trust and multiplicative untrust relations. In the context of software management systems, this corresponds to denying trust to a whole repository, and in turn to all packages with dependencies located in it.

Our approach (conceptually) modifies the standard takes on (un)trust in two ways:

1. *(un)Trust is (un)Trustworthiness of a relation*: (un)trust characterizes a relation, in our application case the access and query operation from a client to a repository for any given package;
2. *Untrustworthiness is qualified by intentionality*: which form of (un)trust is applied, depends on the intention of the client to preserve a local functionality, or its willingness to give it up.

We start by providing informal descriptions of the (un)trust protocols modelled by the logic (`un`)`SecureND` introduced below.

3.1 Trust

We start with basic (positive, informally stated) rules that contextualize our understanding of trust:

- A client queries a repository for a package;
- Packages which can be consistently subsumed under the client's installation profile are trusted;
- A client accessing trustable packages is allowed to install them.

If we consider two basic operations from access control theory like *reading* and *writing*, trust is modelled as an authorization function: a consistently accessible package can be trusted; an accessible and trusted package can be installed, see Figure 5.

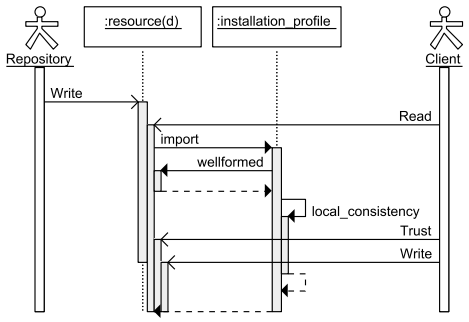

Figure 5: The trust function in a sequence diagram.

3.2 Distrust

Packages conflicting with currently installed ones in the client's installation profile require a conflict resolution strategy. A procedure to distrust a package ϕ from repository under the installation profile of a client is modelled as the refusal to remove currently installed packages. This account of trust denial for the external package requires blocking the content received from the external repository and preserving the originally available functionalities in the installation profile, see Figure 6. The `Block` procedure simulates a strong negation introduction, similar to the abort operation in constructive logic for program actions. It induces rejection of further packages that would depend on ϕ, but not of a further package ψ from the same repository.

3.3 Mistrust

A software package currently installed and conflicting with a desired installation is mistrusted inducing a change in trust behaviour in the installation profile of the client. Under this reading, a procedure to mistrust package ϕ originating in the installation profile of the client expresses the belief (or a measure thereof) that some functionality in the installation profile of such client requires a change in trust attitude in order for a different functionality provided by a conflicting package ψ to be made available. Hence, any trusted information which is directly contradicted by a required resource should be *mistrusted*. Formally, this procedural account of mistrust can be formulated as saying that a trusted resource inconsistent in view of new resources is mistrusted (trust removal) and the previously conflicting resource becomes installable, see Figure 7.

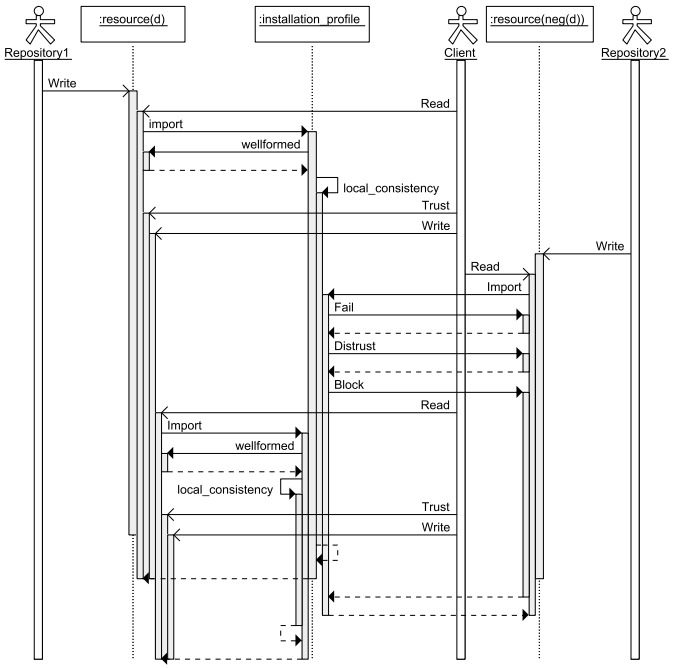

Figure 6: The Distrust function in a sequence diagram.

4 (un)SecureND

The logic `SecureND` was first introduced in [43] to provide a proof-theoretic treatment of trust in the context of access control systems. The basic intuition behind the calculus is the use of a `trust` function to bridge `read` and `write` operations on formulas. In this way, message reading operations including application of trust are guaranteed to preserve consistency when the agents perform writing operations.

The basic application was initially the resolution of problems generated by transitive trust operations, where one wishes to block trust applications among agents where consistency is lost. This protocol was applied in [11] for the first time to the context of software management, where agents are interpreted as software repositories and clients. We offered a trust-based version of the optimization problem from [50], known as the *minimum install problem*: it consists in 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.

In [41] the language has been extended to a negation complete version called

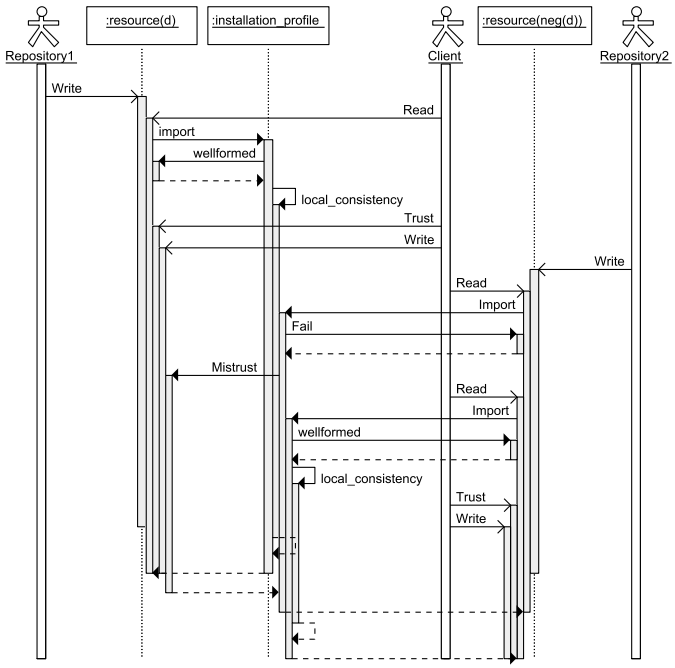

Figure 7: The Mistrust function in a sequence diagram.

(un)SecureND. The present version of (un)SecureND introduces a strict partial ordering on formulas to express package dependency; this is then lifted at the level of sets of formulas representing installation profiles to express rules for their 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:

Definition 6. *(**Syntax of** (un)SecureND)*

$$\begin{aligned}
&\mathcal{S}^{\sim} := \{A \leq B \leq \ldots\} \\
&\phi^S := a^S \mid \neg\phi_i^S \mid \phi_i^S \rightarrow \phi_j^S \mid \phi_i^S \wedge \phi_j^S \mid \phi_i^S \vee \phi_j^S \mid \bot \mid \\
&\qquad Read(\phi^S) \mid Write(\phi^S) \mid Trust(\phi^S) \\
&\Gamma^S := \phi_i^S \mid \phi_i^S < \phi_j^S \mid \Gamma^S; \phi_j^S
\end{aligned}$$

We explain the elements of the language step by step in the following subsections. The language defines a derivability relation between judgements:

Definition 7 (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*.*

The relation of derivability embeds therefore the idea of package dependency, more clearly expressed by the `Dependency Insertion` rule, see below Section 4.1: a judgement expresses package execution, while package dependency alone is introduced in terms of an order relation. A package which does not require any dependency is valid in any installation profile:

Definition 8 (Validity)**.** *An* (`un`)`SecureND` *judgement* $\vdash \phi_i^A$ *says that a package* ϕ_i *from repository* A *can be executed in any profile.*

4.1 Repositories, packages and dependencies

$\mathcal{S}^{\sim}$ is the set of software repositories ordered by $\leq$: this order is determined by dependencies between packages they contain, obtained below as lifting from package dependency $\phi_i^A < \psi_j^B$. A partial order relation $\leq$ over $\mathcal{S} \times \mathcal{S}$ intuitively expresses that dependencies are satisfied across repositories. Note that we preserve the order for logically equivalent repositories $S = S'$, with the understanding that a package dependence can hold between two packages in the same (or in two distinct but logically equivalent) repositories.

Definition 9. $\forall A, B \in \mathcal{S}^{\sim}$*,* $A \leq B$ *if and only if* $\exists \phi_i^A, \psi_j^B$ *such that* $\phi_i^A < \psi_j^B$ *and* $\neg\exists \phi_k^A, \psi_l^B$ *such that* $\psi_l^B < \phi_k^A$*.*

ϕ^S is a meta-variable for formulae, denoting software packages and their logical composition inductively defined by connectives, including operations to read (query), trust (consistency checking) and write (install), with S here a metavariable for elements in $\mathcal{S}^{\sim}$. Each package is indexed by the repository it originates from: ϕ_i^A says that package ϕ_i can be retrieved from repository $A \in \mathcal{S}$. The partial order defined over elements $S \in \mathcal{S}$ 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. The language includes $\bot$ to express conflicts: we formulate $\neg\phi_i^A$ as an abbreviation for $\phi_i^A \rightarrow \bot$, meaning that package ϕ_i^A induces a conflict (possibly within a given installation profile).

By the first clause in Definition 9, $A < B'$ means that some package in A satisfies a dependency for a package in B. By the second clause in Definition 9, our order relation abstracts from the issue of reciprocal dependencies. As noted in [10], 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.

An installation profile Γ^S is the list of all packages sufficient to an access or execution operation on a package $\phi^{S'}$ with $S \leq S'$ and is consistent if it prevents conflicts, i.e. it does not include packages $\phi^S, \neg\phi^S$, or packages ϕ^S, ψ^S such that any dependency of ψ^S is satisfied by a package $\neg\phi^S$. A profile is internally structured to reflect the dependency of packages through the partial order $<$ in $\mathcal{S}^{\sim}$: this is guaranteed by the Profile Construction Rules in Figure 8. We use the meta-theoretical typing declaration $: profile$ to state that a formal expression (ordered set of formulas) on the left-hand side of the colon can be

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

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

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

Figure 8: The System `(un)SecureND`: Profile Construction Rules

considered a valid installation profile. Validity corresponds here to consistency. Profiles are constructed inductively from the empty profile by the first rule. By `Package Insertion`, validly executed packages without dependencies can be added to an empty installation profile. By `Dependency Insertion`, a non-empty installation profile including package ϕ_i^A which consistently admits the execution of a package ψ_j^B allows the insertion of the latter within the profile by preserving the dependency $\phi_i^A < \psi_j^B$. Note that by the first two rules it is not possible to start constructing a profile by having two packages that are in conflict with each other, even if these do not depend from one another. This is obviously a strong requirement, but we assume it here to guarantee consistent construction of profiles by induction. We allow extension of profiles by packages that are not dependent on previous ones, denoted as $\Gamma^S; \Gamma^{S'} = \{\phi_i^S < \ldots < \phi_n^S; \phi_{n+1}^{S'}\}$, by the rule `Profile Extension`. 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 same operation is validated more in general by an application of the Weakening Rule (see Figure 11). It is worth noting that Weakening will preserve profile consistency as it requires additionally an instance of the *trust* rule (see Figure 10).

4.2 Rules for package execution

The operational rules in Figure 9 formulate package execution procedures closed under compositionality by logical connectives. As it is standard in proof-theoretic semantics, the meaning of our binary (connectives) and unary functions is given by a pair of introduction and elimination rules, the former intended to express how the connective is obtained, the latter how it can be dispensed with. Recall that the (generalised) judgement of the form $\Gamma^A \vdash \phi^B$ says that package ϕ from repository B is validly executable within an installation profile with packages coming from repository A. Then the rule `Atom` establishes valid package execution for any package available in the repository from which all packages in the installation profile are obtained, or across any repository for which dependencies are satisfied. This is reflected by the side condition ex-

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

$$\frac{\Gamma^A \vdash \bot}{\Gamma^A \vdash \phi^B}\ \bot, \text{ for any } B \in \mathcal{S}$$

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

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

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

Figure 9: The System `(un)SecureND`: Operational Rules

pressing validity of the rule for any repository $B \geq A$. $\bot$ says that if a profile is inconsistent, any package becomes executable, reflecting a form of *ex falso sequitur quodlibet*: notice how the generalised derivability holds for packages from any repository, irrespective of their dependencies being satisfied. The introduction rule for conjunction $\wedge$-I allows composition of packages from distinct profiles; by the corresponding elimination rule $\wedge$-E, each composing package can be executed under the combined profiles (with $I = \{A, B\}$). The introduction rule for disjunction $\vee$-I says that a combined profile can validly execute any package from each of the composing profiles; by the corresponding elimination $\vee$-E, each package consistently executed under each individual profile can also be executed under the extended profile. $\to$-Introduction expresses inference of a package from a combined profile as inference between packages (Deduction Theorem); its elimination $\to$-E allows to recover such inference as profile extension (Modus Ponens).

4.3 Access Rules

While the previous rules fragment expresses the semantics of rule execution, we need now to equip the system with a set of rules describing valid access, such that execution is guaranteed to preserve the installation profile validity. This fragment of rules is presented in Figure 10. 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`).

$\neg$-distribution is an essential rule to preserve consistency across the positive and negative fragments of the languages: it ensures that if an operation is

$$\frac{\Gamma^A \vdash \neg\mathcal{O}(\phi_i^B)}{\Gamma^A \vdash \mathcal{O}(\neg\phi_i^B)}\ \mathcal{O} \in \{Read, Trust, Write\},\ \neg\text{-distribution}$$

$$\frac{}{\Gamma^A \vdash Read(\phi_i^B)}\ read,\ \text{for any } \psi_i^B \in \Gamma^B \geq \Gamma^A$$

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

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

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

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

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

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

Figure 10: The System `(un)SecureND`: Access Rules

not possible on a package ϕ_i^B under the installation profile Γ^A, then the same operation must be possible assuming a package for a contradictory functionality holds in that same profile. `read` says that from any consistent installation profile Γ^A a package ϕ_i^B can be read, provided its dependencies are satisfied (if any); this is expressed by the side condition which reflects the order relation of repositories.[5] `trust` works as an elimination rule for `read`: it says that if a package ϕ_i^B can be read from an installation profile Γ^A and its inclusion 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 (the function $Write$). `exec` allows to reduce the access process to the execution rules formulated in Figure 9: it says that any package that is safely installed in a consistent profile can be executed in it. Note how in this way execution of non-safely installable packages is prevented.

The following set of rules extend the $Trust$ function by negation, according to the two forms of negated trust from Section 3. The rules for Distrust are

[5]Note that this side condition can be reformulated as a proper premise in the rule if so required.

intended to preserve the current installation profile in view of a conflict with an external package. The Introduction rule for distrust $\texttt{DTrust} - \texttt{I}$ expresses the following principle: a package ϕ_i^B whose reading is inconsistent with the current installation profile Γ^A is untrustworthy, i.e. the rule for `trust` is not applied; note that in this case the rule induces a straightforward rejection of the package. The corresponding elimination rule $\texttt{DTrust} - \texttt{E}$ uses $\rightarrow$-introduction to induce installation of any package ψ_j^C consistent with the resolution of the conflict with ϕ_i^B by blocking its access.

The rules for Mistrust are intended to modify the current installation profile by accommodating the installation of a currently conflicting external package. To do so, the removal of one or more currently installed packages is necessary. The Introduction rule for mistrust $\texttt{MTrust} - \texttt{I}$ expresses the following principle: given a package ψ_i^B whose reading is inconsistent with the current installation profile Γ^A, identify the (set of) package(s) ϕ_j^A which can be removed from the profile Γ^A so that the profile is still a valid one, ψ_i^B consistently extends it, and it makes ϕ_j^A untrustworthy. The corresponding elimination rule $\texttt{MTrust} - \texttt{E}$ expresses the following: given a consistent profile where some package has been removed and a previously inconsistent one ψ_i^B has been added, identify the set of dependencies that the latter needs to satisfy and which can be consistently added to Γ^A to confirm trust of ψ_i^B. This holds for any required dependency in other repositories, as expressed by the side condition that requires checking for any $C \leq B$. 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 current installation profile. Note that both untrust functions are triggered by the querying operation on a repository, hence conflicts are highlighted before installation.

4.4 Structural Rules

Structural rules for (`un`)`SecureND` hold with restrictions, as illustrated in Figure 11. As a result, the system qualifies as substructural, see e.g. [46]. In the present context, these rules illustrate how installation profiles behave with respect to the ordering and composition of new packages.

`Weakening` usually is formulated as to guarantee that consistency of formula derived is not affected by the extension of their assumptions. In the present interpretation though, it is evident that extending installation profiles needs to be guarded against conflicting ones. As a consequence, the rule is constrained by an instance of *trust*: it says that a valid installation of ϕ_i^A in the profile Γ^A is preserved under a profile extension in view of a package ϕ_j^B if and only if such package is trustworthy, i.e. one whose extension of the current installation profile is provably consistent.

`Contraction` normally expresses the principle that copies can be safely ignored, as the information they provide is already available. Under our interpretation, installation profiles are ordered in view of the dependency relations they instantiate, as by Definition 9. Hence, `Contraction` requires a constraint to

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

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

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

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

Figure 11: The System `(un)SecureND`: Structural Rules

preserve such dependencies: it says that a valid installation of ϕ_k^A is preserved when removing one instance of two identical packages $\phi_i^A; \phi_i^B$, possibly from distinct repositories, provided one preserves the package from the higher repository (if one exists) in the order dependency (as illustrated by the side condition $A \leq B$), so as to guarantee any further dependency below is preserved.

`Exchange` holds for set of formulas, where no order is not present and it expresses the principle that formula derivability is preserved across sets of assumptions in which two formulas are swapped. In the present language, the rule is doubly constrained by the ordered structure of installation profiles: 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 (so as to guarantee that there is no hidden order instantiated by their repositories of origin), and more explicitly if there is no involved dependency between them, as illustrated by the side condition $\phi_i \not< \phi_j$.

Finally, the `Cut` rule expresses valid package execution under profile extension, constrained by the preservation of the (downwards) order relation of dependency $A \leq B$: 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. We show here the admissibility of this rule for the most relevant cases:

- for $\phi_i^B \equiv \neg Trust(\phi_j^B)$: consider the first premise of the rule to be the conclusion of a $\texttt{DTrust} - \texttt{I}$ rule, then the whole rule collapses in a form of $\texttt{DTrust} - \texttt{E}$ rule, where ϕ_j^B is any package consistent with the removal of ϕ_i^B;
- for $\phi_i^B \equiv \neg Trust(\phi_i^A)$: consider the first premise of the rule to be the conclusion of a $\texttt{MTrust} - \texttt{I}$ rule, then $\Gamma^A \equiv \Gamma^A \backslash \{\phi_i^A\}; \psi_j^B$ for some package ψ_j^B for which installation is desired; then the second premise of the `Cut`

rule is an instance of $\texttt{DTrust} - \texttt{I}$. The conclusion illustrates then the situation presented in Figure 12, valid for all $C > B > A$.

$$\frac{\Gamma^A \setminus \{\phi_i^A\}; \psi_j^B \vdash \neg Trust(\phi_i^A) \qquad \Gamma^C; \neg Trust(\phi_i^A) \vdash \xi_k^C}{\Gamma^A \setminus \{\phi_i^A\}; \psi_j^B; \Gamma^C \vdash \xi_k^C} \text{ cut}$$

Figure 12: An instance of the Cut Rule

This instance of the rule shows that any package ξ_k^C which is valid under an installation profile $\Gamma^C; \neg Trust(\phi_i^A)$ which mistrusts a given package ϕ_i^A, will be preserved by any consistent extension of the profile $\Gamma^A \setminus \{\phi_i^A\}; \psi_j^B; \Gamma^C$, and this should be preserved for any repository with possible dependencies from the highest repository A in the order.

5 The Mistrusted Uninstall Problem

In this section we illustrate formally how the Mistrusted Uninstall Problem is solved by the logic (un)SecureND. Recall that such problem is formulated in Definition 4 as follows: determine which packages require to be uninstalled from the current profile, given a conflicting package should be installed.

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,

$$\Gamma^A \vdash Write(\phi_m^B)$$

but 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* consists then in determining the minimal set $\Phi^A = \{\phi_i^A \in \Gamma^A \mid \phi_i^A \to \neg\phi_m^B\}$ which should be removed when installing ϕ_m^B. We will call any such package ϕ_i^A a *mistrusted package*. The problem can be further reformulated as that of identifying the minimal set of formulas Φ^A such that for each $\phi_i^A \in \Phi^A$

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

By $\texttt{MTrust} - \texttt{E}$, given any other set of formulas Γ^C required by ϕ_m^B, it follows

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

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

Lemma 1. *If* $\Gamma^A \vdash Read(\phi_m^B) \to \bot$, *then there is* $\Phi^A \subseteq \Gamma^A$ *such that* $\Phi^A = \{\phi_i^A < \ldots < \phi_n^A\} \vdash Read(\phi_m^B) \to \bot$.

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 $\forall \phi_i^A \not< \phi_h^A$, $\phi_h^A \vdash Read(\phi_m^B)$. Hence it suffices to identify the maximal (according to $>$) ϕ_i^A in conflict with ϕ_m^B and to include it in Φ^A together with all packages $\phi_j^A \in \Gamma^A$ such that $\phi_i^A < \phi_j^A$. We will call Φ^A a *maximally mistrusted set.* □

Lemma 2. *Consider a maximally mistrusted $\Phi^A \subseteq \Gamma^A$ and package ϕ_m^B such that $\Gamma^A \vdash Read(\phi_m^B) \rightarrow \bot$. Then $\forall \phi_i^A \in \Phi^A$, $\phi_i^A \not< \phi_m^B$.*

Proof. By Assumption and Lemma 1, $\phi_i^A \vdash Read(\phi_m^B) \rightarrow \bot$. Then (`un`)`SecureND` allows the derivation from Figure 13. Then $\phi_i^A < \neg\phi_m^B$ holds by Dependency Insertion and $\phi_i^A \not< \phi_m^B$ by contraposition.

$$\dfrac{\dfrac{\dfrac{\dfrac{\phi_i^A \vdash Read(\phi_m^B) \rightarrow \bot}{\phi_i^A \vdash \neg Trust(\phi_m^B)}\, DTrust\text{-I}}{\phi_i^A \vdash Trust(\neg\phi_m^B)}\, \neg\text{-distr}}{\phi_i^A \vdash Write(\neg\phi_m^B)}\, write}{\phi_i^A \vdash \neg\phi_m^B}\, exec$$

Figure 13: An instance of derivation in (`un`)`SecureND`.

□

Theorem 1. ***(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 $\{\phi_i^A < \phi_j^A\} \subseteq \Gamma^A$*

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 1 to include all the dependencies in the maximally mistrusted set. The second condition holds from Lemma 2. 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 The Distrusted Uninstall Problem

In this section we illustrate formally how the Distrusted Uninstall Problem is solved by the logic (un)SecureND. Recall that such problem is formulated in Definition 5 as follows: determine which installation are permissible, given a package should not be installed.

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 the system administrator explicitly wants to preserve. We call such a package ϕ_m^B the *distrusted package*. In the calculus, this is expressed by the conclusion of an instance 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. (un)SecureND allows to express this principle as the request to obtain the maximal set of formulas denoted $\{\Psi_i^N\}$ from any repository $N \geq B$ such that

$$\Gamma^A \vdash \neg Trust(\phi_m^B) \to \{\Psi_i^N\}$$

By DTrust-E, this guarantees the right to install any package $\psi_i^N \in \{\Psi_i^N\}$. The first step consists in transforming our problem in a formulation that removes the trust condition.

Lemma 3. $\Gamma^A \vdash \neg Trust(\phi_m^B) \to \psi_i^N$ *iff* $\Gamma^A; \neg\phi_m^B \vdash \psi_i^N$.

Proof. $\to$ By the assumption $\Gamma^A \vdash \neg Trust(\phi_m^B)$ and by consistent distribution of negation we obtain $\Gamma^A \vdash Trust(\neg\phi_m^B)$; similarly, from the premise $\Gamma^A \vdash \neg Trust(\phi_m^B) \to \psi_i^N$ and consistency of negation we get $\Gamma^A \vdash Trust(\neg\phi_m^B) \to \psi_i^N$. Now apply Write to $Trust(\neg\phi_m^B)$ and eliminate the function through exec; by $\to$-E we obtain $\Gamma^A; \neg\phi_m^B \vdash \psi_i^N$.

$\leftarrow$ 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; $\Gamma^A \vdash Trust(\neg\phi_m^B)$ holds 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 $\Gamma^A \vdash \neg Trust(\phi_m^B) \to \psi_i^N$ is obtained by $\to$-I.

□

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

Lemma 4. *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 Atom; 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 5. *$\Gamma^A; \phi_l^N \vdash Write(\phi_i^N)$ iff $(\phi_l^N \not< \phi_m^N \not< \phi_i^N)$ for any package ϕ_m^N and any repository $N \geq A$ such that $\Gamma^A \vdash \neg Trust(\phi_m^N)$.*

Proof. $\rightarrow$ 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 $\phi_l^N \vdash \phi_i^N$ 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.

$\leftarrow$ 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 from Γ^A as by Lemma 4; 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 2. ***(Distrusted Uninstall)*** *Given a package ϕ_m^B such that $\Gamma^A \vdash \neg Trust(\phi_m^B)$, $\Gamma^A \vdash Write(\psi_i^N)$ iff $\phi_m^B \not< \psi_i^N$.*

Proof. From Definition 5, consider the package ϕ_m^B that is distrusted under Γ^A. Then by Lemma 5, any other package ψ_i^N can be installed in Γ^A if and only if ϕ_m^B does not satisfy any direct or indirect dependency for ψ_i^N. □

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

7 Applying the Example

Consider the simple scenario presented in Section 1 where a user aims at expanding an existing installation profile Γ with the following package ad its dependencies (in the following, we remove indices denoting repositories for simplifying reading):

$$\Gamma = \{\ \{libc6, libssl0.9.8\} < tor\ \}$$

Assume the system distrusts the package *libssl*0.9.8 as it is obsolete. The Distrusted Uninstall Problem asks which packages can be further installed in Γ without installing *libssl*0.9.8. Consider now the package *libssl*1.0.0 $\not>$ *libssl*0.9.8, for obvious versioning reasons, then the derivation from Figure 14 holds. In other words, flagging *libssl*0.9.8 as distrustful in our system does not impede the installation of a package *tor* which can be installed independently from it.

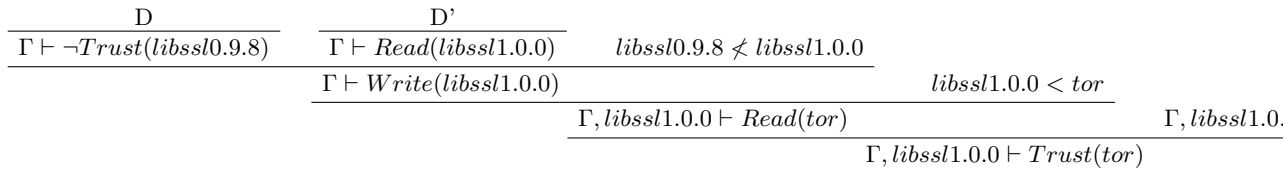

Figure 14: A distrust derivation

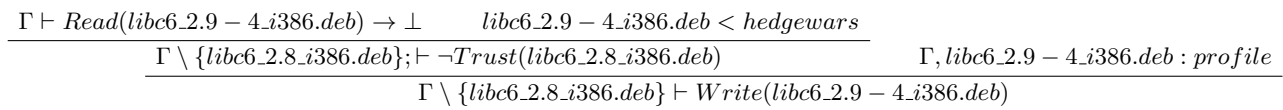

Figure 15: A mistrust derivation

Assume now the user wishes to install an additional package

$$hedgewars > libc6\_2.9 - 4\_i386.deb$$

such that

$$\Gamma = \{libc6 - i686(2.8 \sim 20080505 - 0ubuntu9)\ to\ 2.9 - 4\}$$

and

$$\Gamma \vdash Read(libc6\_2.9 - 4\_i386.deb) \to \bot$$

The package *hedgewars* depends on $libc6\_2.9 - 4\_i386.deb$, but the latter is in version conflict with $libc6 - i686(2.8 \sim 20080505 - 0ubuntu9)\ to\ 2.9 - 4$. Removing the latter proves hard for the presence of other packages like *tor* depending from it. Then the derivation in Figure 15 shows how our mistrust rule manages version upgrade.

8 Coq implementation

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`. It makes it possible to express and prove the lemmas and theorems from sections 5 and 6. Note that the current implementation must be thought of as a proof of realizability in principle of a tool using the proposed library. This is currently at a proof of concept stage and we do not offer details from the usability point of view.

`Coq` is a proof assistant based on the language of type theory and the calculus of inductive constructions. It embeds the formulas-as-types identity originating in the Curry-Howard isomorphism and its computational counterpart, the

proofs-as-programs approach ([22, 32, 39, 19]). Its language is both a pure functional programming language and a type system. A proof assistant is typically used to check proofs, in order to testify their correctness. By the proofs-as-programs formal identity, one can use a proof assistant to test the correctness of a program that has the same logical structure as a given derivation. In `Coq`, `Prop` is the sort or type used for propositions; proof-terms in this type may depend on other terms in `Prop`. An equivalent type is `Set`. The underlying logic for terms is the intuitionistic fragment including conjunction, implication and disjunction, extended with quantifiers and equality.

Theorems are proven by derivation of appropriate sub-goals by applying tactics that use assumptions and provide rules to introduce or eliminate auxiliary propositions. Standard libraries include basic logical notations and properties, basic data types (boolean and natural numbers), operations such as $(+, \times, min)$ and relations such as $(<, \leq)$.

The logic can be axiomatically extended to a classical setting by introducing excluded middle. Additional libraries include, e.g., the rules for algebraic laws or properties of orders, lists, basic functions and properties of lists. Writing of programs uses definition of inductive types, predicates and families, structurally recursive programs, pattern matching.

Our propositions will be typed as `Resources` (packages), defined by logical operations on terms `t` in `Prop`. Our implementation uses the library `Coq.Structures.Orders` to define ordered types, required for the dominance relation between repositories and hence the dependency between packages available in them. It also relies on the `MSets` library for finite modular sets, used for both packages and installation profiles. Equivalence on atomic packages is fully defined in terms of reflexivity, symmetry and transitivity and it is decidable. This typically means that terms are convertible and that a proof of `a` $\equiv$ `b` allows one to substitute `a` for `b` everywhere inside a term.

The different resources are defined in an inductive type:

```
Inductive Resource {A: Type} {S: Type}:
  Type :=
| nd_atom: A -> Resource
| nd_bottom: Resource
| nd_impl: Resource -> Resource ->
    Resource
| nd_and: Resource -> Resource ->
    Resource
| nd_or: Resource -> Resource ->
    Resource
| nd_not: Resource -> Resource
| nd_read: Resource -> Resource
| nd_write: Resource -> Resource
| nd_trust: Resource -> Resource.
```

An installation profile is a list of resources, plus a theorem that specifies when a profile is valid. In such a way, to construct a profile, we will have to supply a list of resources and a proof that this list constitutes a valid profile.

```
Record profile := mkProfile
{ resources: list Resource.t;
  validity: is_valid_aux resources
}.
```

The predicate that specifies whether a list of resources is valid is simple; two succeeding packages in a list are either unrelated or one depends on the other.

```
Function is_valid_aux (l: list Resource.t):
  Prop :=
  match l with
  | [] => True
  | h::t => match t with
    | [] => True
    | h'::t' => ~(Resource.lt h' h) /\
                is_valid_aux t
    end
  end.
```

Equivalence, membership and typability of installation profiles are all defined inductively in terms of the packages they contain and their construction reflects the rules from Figure 8. A judgement in `(un)SecureND` is of the form

```
NDProof: list Resource.t -> Resource.t ->
  Prop
```

where the resource or package `t` is also defined inductively with operations reflecting the rules from Figure 9, 10 and 11.

As an example, this is the Atom rule (renamed `nd_atom_rule` to avoid confusion with the resource `nd_atom`):

```
 nd_atom_rule: forall Ra Rb Pa Pb f,
   typable_profile Ra Pa ->
   typable_profile Rb Pb ->
   repository_leq Ra Rb ->
   typable Rb f ->
   profile_in f Pb ->
   NDProof (profile_merge Pa Pb) f.
```

This covers the rule from the calculus. `Ra` and `Rb` are repositories (in the original rule, these are A and B). `Pa` and `Pb` represent two profiles (Γ^A and Γ^B; the two `typable_profile` rules represent the typing of Γ^A in A and Γ^B in B respectively). The fact that $A \leq B$ is also represented, and the resource `f` typable in `Rb` and part of profile `Pb` represents ϕ_i^B. The conclusion of the rule states that `f` is executable in the combination of profiles `Pa` and `Pb`.

We also need rules for profile construction. Most of these are met through the `validity` theorem; but for dependency insertion, which involves a derivation, we need a special rule:

```
Axiom dependency_insertion:
  forall Pa f, typable Rb f ->
  NDProof (profile_merge Pa
           (singleton_profile f)) g ->
  is_valid_aux (ordered Pa ++ [f; g]).
```

This axiom establishes that for any profile and repository, if a resource `f` is added to a profile `Pa` to execute another resource `g`, then the same profile `Pa` with `f` is valid when `g` is added. This is exactly what the `Dependency Insertion` rule expresses.

Provided the whole calculus is translated into the Coq development, proofs from the former can be transformed into proofs in the latter. An example from the proof of Lemma 1 from Section 5 is given below

```
Lemma lemma1a: forall Ra Rb f1 f2 f3,
repository_lt Ra Rb ->
   Resource.lt f1 f2 -> typable Ra f1 ->
typable Ra f2 -> typable Rb f3 ->
  NDProof (singleton_profile f1)(nd_impl
(nd_read f3) nd_bottom) ->
  NDProof (singleton_profile f2)
(nd_impl (nd_read f3) nd_bottom).
Proof.
  intros.
  apply nd_exec with Ra Rb; sprem.
  apply singleton_profile_typable;
  assumption.
  apply remove_empty.
  apply nd_weakening with Ra Ra;
  sprem.
  apply nd_write_intro with Ra Rb;
  sprem.
  apply nd_read_intro with Ra Rb;
  sprem.
  apply nd_trust_intro with Ra Rb;
  sprem.
  apply nd_read_intro with Ra Rb;
  sprem.
  simpl; trivial.
  apply nd_trust_intro with Ra Ra;
  sprem.
  apply nd_read_intro with Ra Rb;
  sprem.
  simpl; trivial.
Qed.
```

Currently, translating the proofs is still quite labour-intensive. As seen above, rules have many premises, and every proof has to be manually generated. Most of this is quite trivial and can be automated. The `sprem` tactic is a first stab at this, but there is much more that is possible here.

At present, the Coq development consists of a formalisation of the calculus, and a first stab at the proofs. In this way the Coq proofs will be almost direct translations of their paper versions. The idea behind the implementation is that a package installation within a profile can be matched to a corresponding a high-level proof constructed within the syntax of our Coq-base library. Intuitively, the idea is that a user should be provided with the means to present a package intended for installation and the current installation profile of her machine and ask the theorem prover to provide an instance of the validity of a *write* operation for that package under that profile. If the library manages to construct such a proof, the package can be installed, otherwise a warning is issued. The presence of the two negative trust strategies could be implemented by offering the user the option on which to execute: trust removal on currently installed packages, or trust denial of the newly presented conflicitng package. While we are still away from being able to provide such a working tool, future work will be devoted to complete the formalisation of the proofs, and as noted, to write tactics to automate a lot of the administrative legwork.

9 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, grounded on the logic `(un)SecureND`, includes an explicit *trust* function on formulas to guarantee consistency check at each retrieval step (after a *read* function), and 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 and provided an informal explanation of the verification through the Coq proof assistant.

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.

Acknowledgements

This work was done while the first author was a member of the Department of Computer Science at Middlesex University London.

References

[1] P. Abate, R. Di Cosmo, J. Boender, and S. Zacchiroli. Strong dependencies between software components. In *Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement, ESEM 2009, October 15-16, 2009, Lake Buena Vista, Florida, USA*, pages 89–99. ACM / IEEE Computer Society, 2009.

[2] P. Abate, R. Di Cosmo, R. Treinen, and S. 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] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli. Dependency solving: A separate concern in component evolution management. *Journal of Systems and Software*, 85(10):2228–2240, 2012.

[4] A. Abdul-Rahman. *A framework for decentralised trust reasoning.* PhD thesis, Department of Computer Science, University College London, 2005.

[5] A. Abdul-Rahman and S. Hailes. A distributed trust model. In T. Haigh, B. Blakley, M.E. Zurko, and C. Meodaws, editors, *Proceedings of the 1997 Workshop on New Security Paradigms, Langdale, Cumbria, United Kingdom, September 23-26, 1997*, pages 48–60. ACM, 1997.

[6] A.Jøsang, E. Gray, and M. Kinateder. Simplification and analysis of transitive trust networks. *Web Intelligence and Agent Systems*, 4(2):139–161, 2006.

[7] N. Ali, Y.-G. Guéhéneuc, and G. Antoniol. Trustrace: Mining Software Repositories to Improve the Accuracy of Requirement Traceability Links. *IEEE Trans. Software Eng.*, 39(5):725–741, 2013.

[8] C. Artho, K. Suzaki, R. Di Cosmo, R. Treinen, and S. Zacchiroli. Why do software packages conflict? In M. Lanza, M. Di Penta, and T. Xie, editors, *9th IEEE Working Conference of Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland*, pages 141–150. IEEE Computer Society, 2012.

[9] D. Le Berre and A. Parrain. On SAT Technologies for Dependency Management and Beyond. In S. Thiel and K. 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.

[10] J. Boender. Formal verification of a theory of packages. *ECEASST*, 48, 2011.

[11] J. Boender, G. Primiero, and F. Raimondi. Minimizing transitive trust threats in software management systems. In A.A. Ghorbani, V. Torra, H. Hisil, A. Miri, A. Koltuksuz, J. Zhang, M. Sensoy, J. García-Alfaro, and I. Zincir, editors, *13th Annual Conference on Privacy, Security and Trust, PST 2015, Izmir, Turkey, July 21-23, 2015*, pages 191–198. IEEE, 2015.

[12] M. Carbone, M. Nielsen, and V. Sassone. A Formal Model for Trust in Dynamic Networks. In A. Cerone and P. Lindsay, editors, *Int. Conference on Software Engineering and Formal Methods, SEFM 2003.*, pages 54–61. IEEE Computer Society, 2003. A preliminary version appears as Technical Report BRICS RS-03-4, Aarhus University.

[13] J. Carter and A.A. Ghorbani. Towards a formalization of value-centric trust in agent societies. *Web Intelligence and Agent Systems*, 2(3):167–183, 2004.

[14] P.S. Chakraborty and S. Karform. Designing Trust Propagation Algorithms based on Simple Multiplicative Strategy for Social Networks. *Procedia Technology*, 6(0):534–539, 2012. 2nd International Conference on Communication, Computing & Security [ICCCS-2012].

[15] J. Chang, K. Venkatasubramanian, A. West, S. Kannan, B. Loo, O. Sokolsky, and I. & Lee. AS-TRUST: A Trust Quantification Scheme for Autonomous Systems in BGP. In *Trust and Trustworthy Computing: 4th International Conference, TRUST 2011*, volume 6740 of *Lecture Notes in Computer Science*, pages 262–276. Springer Berlin / Heidelberg, 2011.

[16] P.C. Chapin, C. Skalka, and X.S. Wang. Authorization in trust management: Features and foundations. *ACM Comput. Surv.*, 40(3), 2008.

[17] B. Christianson and W.S. Harbison. Why Isn't Trust Transitive? In T.M.A. Lomas, editor, *Security Protocols, International Workshop, Cambridge, United Kingdom, April 10-12, 1996, Proceedings*, volume 1189 of *Lecture Notes in Computer Science*, pages 171–176. Springer, 1996.

[18] S. Clarke, B. Christianson, and H. Xiao. Trust*: Using Local Guarantees to Extend the Reach of Trust. In B. Christianson, J.A. Malcolm, V. Matyas, and M. Roe, editors, *Security Protocols Workshop*, volume 7028 of *Lecture Notes in Computer Science*, pages 171–178. Springer, 2009.

[19] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2–3):95–120, 1988.

[20] R. Di Cosmo and J. Boender. Using strong conflicts to detect quality issues in component-based complex systems. In S. Padmanabhuni, S.K. Aggarwal, and U. Bellur, editors, *Proceeding of the 3rd Annual India Software*

Engineering Conference, ISEC 2010, Mysore, India, February 25-27, 2010, pages 163–172. ACM, 2010.

[21] R. Di Cosmo, S. Zacchiroli, and P. Trezentos. Package upgrades in FOSS distributions: Details and challenges. In T. Dumitras, D. Dig, and I. Neamtiu, editors, *Proceedings of the 1st ACM Workshop on Hot Topics in Software Upgrades, HotSWUp 2008, Nashville, TN, USA, October 20, 2008*. ACM, 2008.

[22] H. B. Curry and R. Feys. *Combinatory Logic, Volume I*. North-Holland, 1958. Second printing 1968.

[23] G. Cvetkovich. The attribution of social trust. In G. Cvetkovih and R. Lofstedt, editors, *Social Trust and the Management of Risk*, pages 53–61. Earthscan, 1999.

[24] G. Cvetkovich and R.E. Lofstedt. Social trust and culture in risk management. In G. Cvetkovih and R. Lofstedt, editors, *Social Trust and the Management of Risk*, pages 9–21. Earthscan, 1999.

[25] T. DuBois, J. Golbeck, and A. Srinivasan. Predicting trust and distrust in social networks. In *PASSAT/SocialCom 2011, Privacy, Security, Risk and Trust (PASSAT), 2011 IEEE Third International Conference on and 2011 IEEE Third International Conference on Social Computing (SocialCom), Boston, MA, USA, 9-11 Oct., 2011*, pages 418–424. IEEE, 2011.

[26] J.A. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[27] T. Grandison and M. Sloman. A survey of trust in internet applications. *Communications Surveys Tutorials, IEEE*, 3(4):2–16, Fourth 2000.

[28] R. Guha, R. Kumar, P. Raghavan, and A. Tomkins. Propagation of Trust and Distrust. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 403–412, New York, NY, USA, 2004. ACM.

[29] P. Herrmann. Trust-based protection of software component users and designers. In P. Nixon and S. Terzis, editors, *Trust Management, First International Conference, iTrust 2003, Heraklion, Crete, Greece, May 28-30, 2002, Proceedings*, volume 2692 of *Lecture Notes in Computer Science*, pages 75–90. Springer, 2003.

[30] H. Hexmoor, S. Rahimi, and R. Chandran. Delegations guided by trust and autonomy. *Web Intelligence and Agent Systems*, 6(2):137–155, 2008.

[31] R. R. Hoffman, J. D. Lee, D. D. Woods, N. Shadbolt, J. Miller, and J. M Bradshaw. The dynamics of trust in cyberdomains. *IEEE Intelligent Systems*, pages 5–11, November/December 2009.

[32] W. Howard. The Formulae-as-Types Notion of Construction. In J. Seldin and J. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[33] A. Ignatiev, M. Janota, and J. 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.

[34] A. Jøsang, S. Marsh, and S. Pope. Exploring Different Types of Trust Propagation. In Ketil Stølen, WilliamH. Winsborough, Fabio Martinelli, and Fabio Massacci, editors, *Trust Management*, volume 3986 of *Lecture Notes in Computer Science*, pages 179–192. Springer Berlin Heidelberg, 2006.

[35] A. Jøsang and S. Pope. Semantic Constraints for Trust Transitivity. In S. Hartmann and M. Stumptner, editors, *APCCM*, volume 43 of *CRPIT*, pages 59–68. Australian Computer Society, 2005.

[36] U. Kuter and J. Golbeck. Using probabilistic confidence models for trust inference in web-based social networks. *ACM Trans. Internet Technol.*, 10(2):8:1–8:23, June 2010.

[37] F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. 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.

[38] S. Marsh and M.R. 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.

[39] P. Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory: Lecture Notes*. Bibliopolis, Napoli, 1984.

[40] J. Mohsen and M. Ester. A Matrix Factorization Technique with Trust Propagation for Recommendation in Social Networks. In *Proceedings of the Fourth ACM Conference on Recommender Systems*, RecSys '10, pages 135–142, New York, NY, USA, 2010. ACM.

[41] G. Primiero. A Calculus for Distrust and Mistrust. In S. Mahbub Habib, J. Vassileva, S. Mauw, and M. 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.

[42] G. Primiero and J. Boender. Managing software uninstall with negative trust. In J.-P. Steghöfer and B. Esfandiari, editors, *Trust Management XI - 11th IFIP WG 11.11 International Conference, IFIPTM 2017, Gothenburg, Sweden, June 12-16, 2017, Proceedings*, volume 505 of *IFIP Advances in Information and Communication Technology*, pages 79–93. Springer, 2017.

[43] G. Primiero and F. Raimondi. A typed natural deduction calculus to reason about secure trust. In A. Miri, U. Hengartner, N.-F. Huang, A. Jøsang, and J. 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.

[44] G. Primiero, F. Raimondi, M. Bottone, and J. Tagliabue. Trust and distrust in contradictory information transmission. *Applied Network Science*, 2:12, 2017.

[45] G. Primiero, F. Raimondi, T. Chen, and R. Nagarajan. A proof-theoretic trust and reputation model for VANET. In *2017 IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2017, Paris, France, April 26-28, 2017*, pages 146–152. IEEE, 2017.

[46] G. Restall. *An Introduction to Substructural Logics*. Routledge, 2000.

[47] P. Sztompka. *Trust: a sociological theory*. Cambridge University press, 1999.

[48] M. Tavakolifard, S.J. Knapskog, and P. Herrmann. Trust transferability among similar contexts. In A.Y. Zomaya and M.Cesana, editors, *Q2SWinet'08 - Proceedings of the 4th ACM Workshop on Q2S and Security for Wireless and Mobile Networks, Vancouver, British Columbia, Canada, October 27-28, 2008*, pages 91–97. ACM, 2008.

[49] T.Dang, Z.Yan, F.Tong, W.Zhang, and P.Zhang. Implementation of a trust-behavior based reputation system for mobile applications. In L.Barolli, F.Xhafa, X.Chen, and M.Ikeda, editors, *Ninth International Conference on Broadband and Wireless Computing, Communication and Applications, BWCCA 2014, Guangdong, China, November 8-10, 2014*, pages 221–228. IEEE, 2014.

[50] 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.

[51] J. Vouillon and R. Di Cosmo. On Software Component Co-installability. *ACM Trans. Softw. Eng. Methodol.*, 22(4):34:1–34:35, October 2013.

[52] Z. Yan and C. Prehofer. Autonomic trust management for a component-based software system. *IEEE Trans. Dependable Sec. Comput.*, 8(6):810–823, 2011.

[53] A. Zeller. Can We Trust Software Repositories? In Jürgen Münch and Klaus Schmid, editors, *Perspectives on the Future of Software Engineering*, pages 209–215. Springer Berlin Heidelberg, 2013.

[54] C.-N. Ziegler and G. Lausen. Propagation Models for Trust and Distrust in Social Networks. *Information Systems Frontiers*, 7(4-5):337–358, December 2005.