
Final accepted version (with author's formatting)

This version is available at: http://eprints.mdx.ac.uk/19194/

Copyright:

Middlesex University Research Repository makes the University's research available electronically.

Copyright and moral rights to this work are retained by the author and/or other copyright owners unless otherwise stated. The work is supplied on the understanding that any use for commercial gain is strictly forbidden. A copy may be downloaded for personal, non-commercial, research or study without prior permission and without charge.

Works, including theses and research projects, may not be reproduced in any format or medium, or extensive quotations taken from them, or their content changed in any way, without first obtaining permission in writing from the copyright holder(s). They may not be sold or exploited commercially in any format or medium without the prior written permission of the copyright holder(s).

Full bibliographic details must be given when referring to, or quoting from full items including the author's name, the title of the work, publication details where relevant (place, publisher, date), pagination, and for theses or dissertations the awarding institution, the degree type awarded, and the date of the award.

If you believe that any material held in the repository infringes copyright law, please contact the Repository Team at Middlesex University via the following email address:

eprints@mdx.ac.uk

The item will be removed from the repository while any claim is being investigated.

See also repository copyright: re-use policy: http://eprints.mdx.ac.uk/policies.html#copy
Minimizing Transitive Trust Threats in Software Management Systems

Jaap Boender, Giuseppe Primiero, Franco Raimondi
Department of Computer Science
Middlesex University, United Kingdom
E-mail: J. Boender—G. Primiero—F. Raimondi@mdx.ac.uk

Abstract—We consider security threats in software installation processes, posed by transitively trusted dependencies between packages from distinct repositories. To analyse them, we present SecureNDC, a Coq implemented calculus using an explicit trust function to bridge repository access and software package installation rights. Thereby, we resolve a version of the minimum install problem under trust conditions on repositories.

I. INTRODUCTION

Trusted interactions between humans and computational systems are at the core of various security applications. Determining the extent and limits of such trust relations is crucial. Moreover, it is not unusual for the actual behaviour of trusted (computational) parties to remain hidden to the human users. This increases risks for both the user and for the stability and security of the system.

Recently, a number of different formal approaches combining security and reputation models with trust in uncertain environments, autonomous systems and social networks have been presented that study trust propagation, trust interference and distrust blocking, see e.g. [GM82], [CNS03], [GKRT04], [ZL05], [MD05], [JP05], [CCX09], [KG10], [CVW+11], [DGS11]. Trust has been applied in many Internet-based services, see e.g. [GS00] for an overview, with specific applications to component-based systems [Her03], [YP11], software management systems related to security and reputation [BDS11], accuracy [AGA13], trust transferability in context-aware [TKH08] and mobile applications [DYT+14], and epistemic reliability [Zel13]. A problem related to (on- and off-line) trust-based interactions is their (possibly unintentional) transitive nature: “Agent A trusts agent B, and agent B trusts agent C. Therefore, agent A trusts agent C”. This problem [CH96] is considered explicitly in the context of logical models for trust management, e.g. Datalog [LM03] and Cassandra [BS04], with a standard solution being that of fixing a bound to transitivity delegation depth, see e.g. [CSW08]. The formalism of subjective logic is used in [JMP06] to define different trust transitivity operators to account for uncertainty and individuality of beliefs in a way compatible with standard logic and probability calculus; in [JAM12] trust transitivity is instead based on conditional belief reasoning similarly to analysing competing hypotheses and weighing evidence to draw conclusions about the trust target. The subjective logic approach explicitly deals with the resolution of uncertainty related to subjective opinions. Subjectivity is also approached as a problem in trust management and recommendation, see [HBPB09]. Besides the probabilistic approach, crisp and fuzzy evaluation methods have also been presented, see e.g. [GH06], [SSDC07], [NBL10].

Software packages management and installation processes offer a novel field to explore trust transitivity. Consider the following example. A user interacts with a software package system, like apt-get on a Debian platform; through such a system, the user is able to install or remove applications, each operation requiring preservation of validity for the current installation profile, i.e. one which meets all dependencies and avoids conflicts for all the packages installed. Then the following problem can be formulated (see [TSJL07]):

Definition 1 (Minimum install problem). Determine the way to install a new package \( p \) such that the minimal amount of dependencies is satisfied and conflicts are avoided to guarantee a valid installation profile.

This problem has obvious implications for security: to be aware of the required dependencies means to be able to monitor and prevent the installation of undesired packages which can turn out to be malwares or trojans. In [TSJL07], the Minimum Install Problem is formulated in view of an objective function aimed at minimizing the number and size of packages required and delivered for obtaining a valid profile under installation. In the present paper we introduce a version of the Minimum install problem involving transitively trusted packages, retrieved from repositories for installation. Repositories are of two sorts: those used by default by the OS, e.g. the official repositories where the basic system and the authorised upgrades are available; and those that the user needs to explicitly authorize, e.g. to install and then automatically update packages that are not included in the official version of the system. This difference in implicit vs. explicit authorization is reflected in the way packages satisfy dependencies during an installation process. Unfortunately, the trust chain of required packages does not terminate after one step. Both official and third-party repositories might be using software from further repositories, automatically adding them to the user’s system. In most cases, these are bona fide services. Sometimes, though,
downloading a package originating from a third-party source offers a backdoor to the user’s system, becoming a security threat. Cautious systems require super-user privileges, but they hardly allow a strict control of all transitive trust relations. In general, the user should perform the installation that requires the minimum amount of such transitive dependencies because each of those represents a threat. The minimum install problem can now be reformulated under transitive trust relations:

Definition 2 (Minimally Trusted Install Problem). Determine the way to install a new package \( p \) such that the minimal number of transitive trusted packages is required from external repositories to guarantee a valid installation profile.

This version of the minimum install problem requires that a valid installation profile is obtained by installing all dependencies required by \( p \) with the minimal number of new transitive trusted packages. The approach in [TSJL07] which encodes propositional constraints over packages is not easily extended to our version of the Minimum Install problem: we wish not only to have a guarantee that the chosen path of dependencies is the shortest one, but also that such path is as secure as one where all required packages are located on a non-transitively trusted repository. Our aims are:

1) to offer a provably correct calculus for operations on software packages, including trust authorization from third-party repositories;
2) to prove the equivalence of an installation under transitively trusted packages from external repositories to one under import on a local profile;
3) to define an algorithm that, taken each possible installation path for a package, it returns an ordering on the basis of the number of transitive trust operations to be eliminated, so as to choose the path with the minimal number of such operations.

Our starting point is SecureND, a natural deduction calculus introduced in [PR14] for an access control model with an explicit trust function. It resolves the problem of trust transitivity by requiring explicit resources import in the installer’s local profile. The cut elimination theorem proven for SecureND by requiring explicit resources import in the installer’s local profile introduced in [PR14] for an access control model with an extended profile. The syntax of SecureND is defined by the following alphabet:

\[
S := \{ A, B, \ldots \} \\
\phi^S := a^A | \phi^S \rightarrow \phi^S | \phi^S \land \phi^S | \phi^S \lor \phi^S | \text{Read}(\phi^A) | \text{Write}(\phi^A) | \text{Trust}(\phi^A) \\
\Gamma^A := \phi^A_1, \ldots, \phi^A_n; \\
\]

where \( S \) is a set of Software Repositories; \( \phi^A \) is a software package inductively defined from atomic formulas typed by its originating repository \( A \in S \); \( \Gamma^A \) expresses an installation profile that includes software packages from a repository \( A \).

Rules for logical connectives, illustrated in Figure 1 are used for modular package construction. Access functions are understood as follows: Read is a query of a software package in a repository; Write is the installation operation; Trust is the addition of a package to the current installation profile under consistency constraints. The role of this operation is to bridge reading privileges (querying) to writing ones (installing). A SecureND-formula \( \Gamma^A \vdash \phi^B \) states that under installation profile \( \Gamma \) containing software from repository \( A \), some access operation on a package \( \phi \) from repository \( B \) is valid. Profile validity is expressed by context construction operations that preserve consistency. For the base case: \( \Gamma^A, \vdash \psi \) iff \( \Gamma^A \neq \emptyset \) and \( \forall \psi^S \in \Gamma^A, S = A \), i.e. the profile is well-formed if it is not empty and if every package it contains is valid in the repository typing the profile. For the step case: profile extension corresponds to import of a package within a profile. If the import concerns a single package from the same repository, extension is expressed as \( \Gamma^A, \phi^A \); profile extension by a package from a distinct repository is expressed as \( \Gamma^A, \phi^B \), i.e. it requires construction of an extended profile. A profile \( \Gamma^A \) satisfies a dependency clause \( \phi_1 \lor \ldots \lor \phi_k \) for a package \( \psi^B \) iff at least one \( \phi_i \) is present in \( \Gamma^A \). A profile \( \Gamma^A \) satisfies a conflict clause \( \phi_i \) for a package \( \psi^B \) iff \( \phi_i \notin \Gamma^A \). In the following, the possibility that a package may depend from another one within the same repository is not explicitly accounted for: this problem, dealt with in [MBC+06] for open source software repositories, is simplified by considering the dependency as always instantiated between different repositories (eventually, a repository can be considered as a singleton when needed). Moreover, we further simplify the real-case scenario by assuming always linear dependencies between repositories, i.e. we ignore the possibility that a package \( \phi^A_i \)
depends from a package \( \psi^B \) while there is also a package \( \psi^B \) depending from \( \phi^A \). Finally, repositories (and hence therein contained packages) are positioned in a partial order so that each dependency is expressed in function of a higher position in the dominance relation.

**Definition 4 (Dependent Repositories).** A relation \( \leq \) over \( S \times S \) is a partial order such that \( S \leq S' \) iff there is at least one package \( \phi^{S'} \) that has a dependency relation on a package \( \psi^S \), and no dependency exists in the other direction.

Our system aims at implementing a strong security policy for the transitivity of operations across different repositories, by requiring explicit trust on imported packages. Given \( S < S' < S'' \), if \( S \) reads a package from \( S' \), and \( S' \) is allowed to install content from \( S'' \), then \( S \) should be allowed to install from \( S'' \) if and only if \( S' \) trusts content from \( S'' \). In other words, when an installation profile accesses a package from a repository, we do not extend the privileges directly to installing rights even if the dependency is in favour of the accessed repository. This is summarized in the following list of valid operations according to the dominance relation among repositories:

- \( \Gamma^S \vdash \text{Read}(\phi^S) \) and \( \text{Write}(\phi^S) \) hold;
- \( \Gamma^S \vdash \text{Read}(\phi^S \geq S) \) holds;
- \( \Gamma^S \vdash \text{Write}(\phi^{S'} \geq S) \) holds under trust;
- \( \neg \exists S') \Gamma^S \vdash \text{Trust}(\phi^{S'}) \).

To make a preliminary comparison with a real case scenario consider the Debian distribution and its apt-get software management system. It implements the ordering \( \text{main} \leq \text{non-free} \leq \text{contrib} \) where:

- the main repository contains all free packages;
- the non-free repository contains all packages with onerous license conditions that need packages in main;
- and the contrib repository contains all freely licensed packages that depends from packages in non-free.

In our language, we express the validity of an operation on a package available from a given repository, under a certain installation profile containing all packages available from a given repository (possibly the same). Possible instances of valid expressions are:

- \( \Gamma^\text{main} \vdash \phi^\text{main} \), which expresses an operation on a package in main, performed from an installation profile that contains software from the same repository;
- \( \Gamma^\text{main} \vdash \phi^\text{non-free} \), which expresses an operation on a package in non-free, performed from an installation profile that contains software from main;
- \( \Gamma^{\text{non-free}} \vdash \phi^\text{main} \), which expresses an operation on a package in main, performed from an installation profile that contains software from non-free.

In the actual case of the Debian distribution, installing a package from non-free in a profile typed by main requires authorizations, while an installation of a package from main in a profile typed by non-free is admissible by default. Because of the transitivity of the dominance relation among repositories (if \( S < S' \) and \( S' < S'' \), then \( S < S'' \)), as a side effect we also have that if \( \Gamma^{\text{contrib}} \vdash \text{Write}(\psi^\text{non-free}) \), and \( \psi^\text{non-free} \vdash \text{Read}(\phi^\text{main}) \) then \( \Gamma^{\text{contrib}} \vdash \text{Read}(\phi^\text{main}) \), by default. For querying operations these is usually trivial and not harmful, but it leaves space to possible threats especially for installing operations. As such, it represents a behaviour that one might want to restrict, in particular in those cases where the repository that is transitively trusted on installing a package is not main, but a third-party one.

III. THE IMPLEMENTATION SecureNDC

**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 identity ([CF58], [How80], [ML84], [CH88]). 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 formal identity underlying proofs and programs, one can use a proof assistant to test the correctness of a program that has the same logical structure of a given derivation. **Coq** uses the sort **Prop** for propositions (equivalent to **Set**); only terms in this sort (proof-terms) may depend on other terms in **Prop**. The underlying logic for terms is the intuitionistic
fragment \{\land, \rightarrow, \lor\}, extended to quantifiers and equality. Goals are reached by derivation of appropriate sub-goals by applying tactics that use assumptions and provide rules to introduce or eliminate auxiliary propositions (different for each logical form available). 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, \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. Programs use definitions of inductive types, predicates and ties of orders, lists, basic functions and properties of lists. Libraries include e.g. the rules for algebraic laws or properties of orders, lists, basic functions and properties of lists.

The full Coq-implementation SecureNDC is freely available as [BPR15]. It uses:

- the Coq.Structures.Orders library to define ordered types, required for the dominance relation between installation profiles available from repositories, and hence the dependency between packages;
- the MSets library for finite modular sets, used for both the sets of packages and installation profiles.
- equivalence on resources is fully defined in terms of reflexivity, symmetry and transitivity and is hence 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.

In the following, read operations are performed by user installation profiles who are granted access to software packages from repositories; trust on a package is the explicit inclusion of the package from the corresponding repository (e.g. by means of a trusted key) in the installation profile; write is package installation within the profile. Repositories are list of software packages, with axioms for equality and decidability, ordered by a dominance relation mimicking dependency. Each software package is a resource belonging to a repository, closed under equivalence and logical connectives:

\[
\begin{align*}
\text{Inductive Resource} &\ (A: \text{Type}) \ (S: \text{Type}): \text{Type} := \\
&| \text{nd_atom} : A \rightarrow \text{Resource} \\
&| \text{nd_impl} : \text{Resource} \rightarrow \text{Resource} \rightarrow \text{Resource} \\
&| \text{nd_and} : \text{Resource} \rightarrow \text{Resource} \rightarrow \text{Resource} \\
&| \text{nd_or} : \text{Resource} \rightarrow \text{Resource} \rightarrow \text{Resource} \\
&| \text{nd_read} : \text{Resource} \rightarrow \text{Resource} \\
&| \text{nd_write} : \text{Resource} \rightarrow \text{Resource} \\
&| \text{nd_trust} : \text{Resource} \rightarrow \text{Resource}.
\end{align*}
\]

An installation profile is a set of packages typed by a repository; a profile is typable if all its packages are typable from a repository, and well-formed if not empty (typability is a decidable property):

\[
\begin{align*}
\text{Module Profile} &\ (R: \text{REPOSITORY}) \ (A: \text{ATOM} \ R). \\
\text{Module E := Resource} &\ (R) \ (A). \\
\text{Include WSetsOn E.} \\
\text{End Profile.}
\end{align*}
\]

\[
\begin{align*}
\text{Parameter typable} : \\
\text{Repository} &\rightarrow \text{P.E.} \rightarrow \text{Prop.}
\end{align*}
\]

\[
\begin{align*}
\text{Definition well_formed} &\ (P: \text{P.t}): \text{Prop} := \\
&\text{forall f, In f P} \rightarrow \text{typable R f.}
\end{align*}
\]

A SecureNDC-formula NDProof (Pa::Pb::nil) b expresses that a software package \( b \) typable in \( P \) is accessible from a profile \( Pa \) typable by repository \( Ra \). Figure 2 offers the inductive definition of the package constructions under logical connectives. nd_atom_mess defines the behaviour of an atomic package with respect to profiles and repositories: it types an individual package \( b \) typable in \( Rb \) as available to all profiles \( Pa \) in repository \( Ra \) above in the dominance relation when accessing profile \( Pb \). nd_and_intro constructs packages \( f_1, f_2 \) from distinct profiles \( Pa, Pb \) typed from different repositories, respectively \( Ra, Rb \); by each of \( \text{nd_and_intro} \_l \) and \( \text{nd_and_intro} \_r \), each component \( f_1 \) or \( f_2 \) of a modular package can be obtained from the combined repositories. Each \( \text{nd_or_intro} \_l \) and \( \text{nd_or_intro} \_r \) allows to access a package \( f_1 \) (respectively \( f_2 \)) within an extended profile \( Pa::Pb::nil \) to form an extended package \( \text{nd_or} \_f1 f2 \). The corresponding elimination \( \text{nd_or_elim} \) allows to operate on a new package \( f \) from the current extended profile \( Pa::Pb::nil \) when each profile allows to operate on \( f \) individually. nd_impl_intro and \( \text{nd_impl_elim} \) establish packages dependency within a profile authorised downward in the dominance relation between repositories. nd_read_intro says that a package \( f \) that can be obtained by repository \( Rb \) is readable under a profile \( Pa \) in repository \( Ra \), i.e. going up in the dominance relation. \( \text{nd_trust_intro} \) says that a package \( f \) that can be read under \( Pa \) and added to it, is trustable under \( Ra \). Trusting a package can be interpreted as extending the current installation profile with the package’s source. \( \text{nd_write_intro} \) says that a package \( f \) accessible in \( Rb \) and that is read and trusted under \( Pa \), can be written (installed) under that profile.

If under a well-formed installation profile \( Pa \) in \( Ra \) one wants to deduct a constant \( b \) available from a profile \( Pb \) in \( Rb \), one needs to import the latter profile in the former. The relation between repositories is expressed by properties of the dominance relation:

\[
\begin{align*}
\text{Parameter typable} \_l \_read : &\forall f \ R \ P \ P', \\
\text{typable} \_l \_profile &\ R \ P \rightarrow \text{typable} \ R \ f \rightarrow \\
\text{NDProof} \ P \ P' \ (P::nil) \ (\text{nd_read} \ f).
\end{align*}
\]

\[
\begin{align*}
\text{Parameter typable} \_l \_write : &\forall f \ R \ P, \\
\text{typable} \_l \_profile &\ R \ P \rightarrow \text{typable} \ R \ f \rightarrow \\
\text{NDProof} \ (P::nil) \ (\text{nd_write} \ f).
\end{align*}
\]

\[
\begin{align*}
\text{Parameter typable} \_2 \_read : &\forall f \ R \ R' \ P, \\
\text{typable} \_2 \_profile &\ R \ P \rightarrow \text{typable} \ R' \ f \rightarrow \\
\text{Repository} \ lt \ R' \rightarrow \\
\text{NDProof} \ (P::nil) \ (\text{nd_read} \ f).
\end{align*}
\]

\[
\begin{align*}
\text{Parameter typable} \_2 \_write : &\forall f \ R \ R' \ P, \\
\text{typable} \_2 \_profile &\ R \ P \rightarrow \text{typable} \ R' \ f \rightarrow \\
\text{Repository} \ lt \ R' \rightarrow \\
\text{NDProof} \ (P::nil) \ (\text{nd_write} \ f).
\end{align*}
\]

\[
\begin{align*}
\text{Parameter typable} \_2 \_profile : &\forall f \ R \ P, \\
\text{typable} \_2 \_profile &\ R \ P \rightarrow \text{typable} \ R \ f \rightarrow \\
\text{Repository} \ lt \ R' \rightarrow \\
\text{NDProof} \ (P::nil) \ (\text{nd_profile} \ f).
\end{align*}
\]

\[
\begin{align*}
\text{Parameter typable} \_2 \_profile : &\forall f \ R \ P, \\
\text{typable} \_2 \_profile &\ R \ P \rightarrow \text{typable} \ R \ f \rightarrow \\
\text{Repository} \ lt \ R' \rightarrow \\
\text{NDProof} \ (P::nil) \ (\text{nd_profile} \ f).
\end{align*}
\]
Parameter typable_3_write: forall f R R' P,
    typable_profile R P -> typable R' f ->
    (NDProof (P::nil) (nd_write f) <->
     NDProof (P::nil) (nd_read f) /
     NDProof (P::nil) (nd_trust f)).

Assuming Ra<Rb in a ‘write-down’ policy, when installing from Ra, all sets of packages in Rb are trusted; when installing under Rb, profiles in Ra are not trusted by default and this has to be formulated explicitly in the calculus. In general, it is not possible within this system to install packages from any repository. If the import of a package within the installation profile is obtained by accessing a repository upwards in the dominance relation, then one is required to trust packages. This way also unintentional transitivity is restricted: a package is trustable iff it can be made explicitly part of one’s installation profile.

IV. TRUSTED INSTALLATIONS

When packages are available from different repositories than the one for the current profile, their installation requires an import operation that makes the package part of the installation profile, preserving its well-formedness.

Axiom nd_import: forall f,
    NDProof ('Pa::nil) (nd_read f) ->
    typable ('Rb) f ->
    typable_profile Ra (P.add f ('Pa)).

This operation represents a security threat. When working with reliable repositories, one should be able to prove that operations under import are equivalent to those where all packages are included in the current installation profile. In a natural deduction calculus, import corresponds to an instance of a cut rule and the required good behaviour corresponds to proving a cut-elimination theorem: any derivation step containing a cut-rule can be eliminated without loss of information. In SecureNDC, we provide a general Cut-Elimination theorem that depends on nd_import. The theorem says that any package installation making use of either packages below in the dominance relation or explicitly trusted will be equivalent to an operation where all the packages required by the current installation profile are safely included in the installation profile.

Theorem 1 (Cut-Elimination Theorem). Any SecureNDC derivation can be transformed into another one with the same final NDProof without nd_import iff

1) either the repository typing the installation profile is dominating the dependency relation for any package required by the installation operation;

2) or trusted nd_import is explicitly granted by the current installation profile on the upward domination relation for each package f involved in the installation operation.

Proof. For point 1), one needs to show that import is possible downwards in the dominance relation among repositories; when performing under repository Ra, a request to add a protocol from repository Rb preserves well-formedness of Pa;
then any derivation with the downward import (NDDCProof) is equivalent to one without import (NDProof):

Lemma nd_import_write_down:
forall f, in f ('Pb) ->
NDProof ('Pa::nil) (nd_write f) ->
typable_profile Ra (P.union ('Pa) ('Pb)).

Inductive NDDCProof: list P.t -> P.E.t -> Prop :=
| dc_normal_proof: forall D f,
  NDProof Ra Rb Pa Pb D f ->
  NDDCProof D f
| down_cut: forall f x,
typable ('Rb) f -> typable ('Rb) x ->
NDDCProof ('Pb::nil) f ->
NDDCProof (P.add f ('Pb)::nil) x ->
NDDCProof ('Pa::'Pb::nil) x.

Theorem down_cut_elimination: forall P f,
NDDCProof P f -> NDProof Ra Rb Pa Pb P f.

Point 2) reflects the case when the download and install operations require trusting upward in the dominance relation between repositories. This corresponds to two formulations of the upward import. In the first case, it means one needs to show that under Pb, any import of a package from Pa preserves wellformedness; in the second case, it requires showing that under Pb, any import of a profile Pa similarly preserves wellformedness. In both cases, one wants to prove that any derivation with the upward import (NDUCProof) is equivalent to one without (NDProof).

Inductive NDUCProof: list P.t -> Prop :=
| uc_normal_proof:
  forall P f, NDProof Ra Rb Pa Pb P f ->
  NDUCProof P f
| up1_cut: forall f x,
typable Ra f -> typable ('Rb) x ->
P.In f ('Pa) -> NDUCProof ('Pb::nil) f ->
NDUCProof ('Pa::'Pb::nil) x ->
NDUCProof ('Pa::'Pb::nil) x
| up2_cut: forall f x,
typable ('Rb) f -> typable Ra x ->
P.In f ('Pb) -> NDUCProof ('Pb::nil) f ->
NDUCProof ('Pa::singleton f::nil) x ->
NDUCProof ('Pa::'Pb::nil) x.

Theorem up_cut_elimination: forall P f,
NDUCProof P f -> NDProof Ra Rb Pa Pb P f.

Cut-elimination allows to prove normalization of any read-write operation to one where trust-attributes are guaranteed. Under explicitly trusted repositories, an install operation is as safe as one that requires no imports of packages from other repositories.

V. MINIMAL TRUST

In this section we make use of the library defined above to offer a solution for the Minimally Trusted Install Problem. Minimizing the transitive applications of trusted packages typed by repositories means to increase significantly the control over resources and the security of the system during installation operations. Resolving the problem of accepting a minimal amount of transitively trusted packages is now formulated in terms of the number of import applications, equivalent to determining the (minimal) number of steps required to obtain a normalized SecureNDC derivation, i.e. one that satisfies the Cut-Elimination Theorem. To this aim, we are interested in calculating recursively the number of trust operations involved by each derivation. To compute the number of such instances, we present in Figure 3 an algorithm that calculates recursively for each construction a value extracted from the number of applications of the import operation that require a nd_trust_intro rule.

For any profile Pa assumed well-formed by H1, and any atomic package fb assumed in profile Pb by H2 and accessible from Pa, the function returns a null trust value. For profiles Pa and Pb and respective operations H1 and H2 for packages f1 and f2, each having a trust function value n and m, TrMinInst calculates total values according to the various connectives: and_intro sums values n,m; by and_elim, any operation using as an assumption an operation with trust value x in an and-elimination rule, will have also value x; or_intro_l and or_intro_r take the value x used in construction by disjunction; or_elim sums the maximum value n respectively m of two operations with the value x of a further package operation that can be obtained by either of the two; impl_intro takes the trust value of the antecedent in the package operation inducing the consequent; impl_elim considers the trust value n of the operation to obtain the implication, the trust value m of the operation to obtain the consequent and takes the minimum of those in the final operation importing the antecedent in the profile; for read_intro, it just considers the value of the reading profile, adding nothing; for trust_intro, it adds one to the value of the currently using profile (thus effectively increasingly the overall value); for write_intro, it adds the value of trusting the installed package to the overall value of the profile under which installation is performed. The application of a trust rule matches a derivation in which a cut is executed when normalizing. Accordingly, the value of the TrustMin function will give the number of required imports, increasing when these are executed upwards. Given the proven normalization by Theorem 1, we can therefore offer a translation of Definition 2 from Section I in the context of SecureNDC as follows:

Definition 5 (Minimally Trusted Installation Problem). Given profile Pa typed in repository Ra and package fb in profile Pb, obtain (NDUCProof(Pa::Pb::nil) nd_write fb) such that the number of instances of nd_import on Pb to be eliminated to obtain a corresponding NDProof is minimal.

By the Theorem up_cut_elimination we know that such a reduction is possible in general, hence the calculus SecureNDC guarantees that by nd_import operations, trusted installations are possible. By the function TrMinInst
Given multiple configurations of dependency satisfaction under which a package \( fb \) could be installed, the user is now in a position to score all possible valid installation profiles according to the output of \( \text{TrMinInst} \), i.e. in view of the number of required trusted import relations to be eliminated. Hence, the installation path requiring the minimal number of such transitive trust operations is chosen, minimizing the risks for security and stability.

We have presented an implementation in the theorem prover Coq of the typed natural deduction calculus SecureND. The calculus and its implementation embed trust as an explicit attribute of installation profiles. The presented Minimally Trusted Install Problem has an input space in terms of trusted packages. By the structure of our typed calculus, packages reveal information about the originating repositories, hence this second parameter can also be extracted. We focused on cut-elimination and illustrated its meaning for trust-based operations in software management systems. The problem is resolved by an optimization algorithm which returns the shortest path in terms of external packages to be transitively trusted, a result obtained by computing the minimal number of cut-rule applications. Future work shall proceed in several directions. The current proposal is in need of evaluation. This can be done by building a certified functional program in e.g. Objective Caml, Haskell or Scheme extracting it from the Coq implementation of SecureND, to execute any of the routines allowed by this calculus and possibly test the feasibility of the \( \text{TrMinInst} \) algorithm in a real-case scenario. A different example to consider is offered by the case of root certificates: a website often does not operate its own root certificate authority; instead its operation related to the signing of identity certificate can be affirmed and trusted by a different, intermediate authority, whose operations are in turn affirmed and trusted by the root Certificate Authority. Also in this case, a cautious user would want to authorize the minimal number of intermediate authorities and possibly have a guarantee that the result is equivalent to adding directly a root certificate. We also plan to focus on the trusted uninstall operation, where removing a package is considered trusted if it is proven to preserve consistency of the installation profile. An extension of the calculus in this direction can be explored in terms of distrust propagation.

**REFERENCES**


