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 and autonomous systems have been presented that include trust propagation, trust interference and distrust blocking, see e.g. [GM82], [CNS03], [GKRT04], [ZL05], [MD05], [JP05], [CCX09], [CVW+11]. Trust has been applied in many Internet-based services, see e.g. [GS00] for an overview, with specific applications to component-based systems [YP11] and software management systems, e.g. related to security and reputation [BDS11], accuracy [AGA13] and epistemic reliability [Zel13]. A problem related to (on- and off-line) trust-based interactions is that of their transitive nature and their possibly unintentional results.

> "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]. Transitivity (without trust) has been considered also in the context of software management systems. 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 the current installation profile:

Definition 1 ((Valid) Installation Profile)**.** *The set of software packages installed on a machine is called the installation profile of that machine. A valid installation profile is one which meets all dependencies and avoids conflicts for all the packages installed.*

Then the following problem can be formulated (see [TSJL07]):

Definition 2 (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 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 3 (Minimally Trusted Install Problem)**.** *Determine the way to install a new package p such that the minimal number of transitively 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 transitively 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. A different example of a chain of trust relations implementing transitivity 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. In the following we will focus on the context of software management systems only, and leave the analysis of this different case to future research.

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 satisfies point 2) above. In the present contribution, we introduce SecureNDC, which implements SecureND in the language of the theorem prover `Coq`, to satisfy proint 1) (validate correctness of the logic underlying SecureND) and 3) (formalize the algorithmic solution to the Minimally Trusted Install Problem) above. This is, to our knowledge, the first model of transitive trust relations in the context of software management systems that uses a proof-theoretical approach and is translated to and validated by a tool like `Coq`, representing an opportunity for other research to build upon.

The rest of this paper is structured as follows. In section II we briefly overview the principles of the calculus SecureND from [PR14]. In Section III we draw a first comparison to a simplified real case scenario. In section IV we give a brief overview of the `Coq` system and of the libraries required for our implementation. We present the inductively defined set of rules of the calculus SecureNDC, with an interpretation that fits the scenario of software management for security threats under transitive trust. In section V we offer an interpretation of trusted installation via the structural properties of the system. In section VI we analyse the Minimally Trusted Install Problem, offering the algorithm computing for each installation path the maximal number of trust relations to be eliminated.

II. The Calculus SecureND

SecureND, introduced in [PR14], is a typed natural deduction calculus designed for trusted access control on resources. We offer here an interpretation modelled after the software management scenario.

Definition 4 (Syntax)**.** *The syntax of* `SecureND` *is defined by the following alphabet:*

$$\begin{array}{l}\mathcal{S} := \{A, B, \dots\} \\ \phi^S := a^A \mid \phi_1^A \rightarrow \phi_2^A \mid \phi_1^A \wedge \phi_2^A \mid \phi_1^A \vee \phi_2^A \mid \\ Read(\phi^A) \mid Write(\phi^A) \mid Trust(\phi^A) \\ \Gamma^A := \{\phi_1^A, \dots, \phi_n^A\};\end{array}$$

where $\mathcal{S}$ is a set of Software Repositories; ϕ^A is a software package inductively defined from atomic formulas typed by its originating repository $A \in \mathcal{S}$; Γ^A expresses an installation profile that includes software packages from a repository A.

In the present context, we understand *Read* as query of a software package and *Write* as the installation operation. An additional access operation *Trust* is defined on packages and understood as 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 Γ containing software from repository A, some access operation on a package ϕ from repository B is valid. Profile validity is expressed by context construction operations that preserve consistency. For the base case: $\Gamma^A; \cdot \vdash wf$ 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 Γ^A, ϕ^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 Γ^A satisfies a dependency clause $\{\phi_1 \vee \ldots \vee \phi_k\}$ for a package ψ^B iff at least one ϕ_i is present in Γ^A. A profile Γ^A satisfies a conflict clause ϕ_i for a package ψ^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 ϕ_1^A depends from a package ψ_1^B while there is also a package ψ_2^B depending from ϕ_2^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 5 (Dependent Repositories)**.** *A relation $\leq$ over $\mathcal{S} \times \mathcal{S}$ 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 ψ^S, and no dependency exists in the other direction.*

Reading and writing are valid operations within an installation profile for packages originating from the same repository. Our system aims at implementing a stronger security policy for the transitivity of operations across distinct 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 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:

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

III. A simple example

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 $main \leq non-free \leq contrib$ 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:

- $\Gamma^{main} \vdash \phi^{main}$, which expresses an operation on a package in *main*, performed from an installation profile that contains software from the same repository;
- $\Gamma^{main} \vdash \phi^{non-free}$, which expresses an operation on a package in *non-free*, performed from an installation profile that contains software from *main*;
- $\Gamma^{non-free} \vdash \phi^{main}$, 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 $\Gamma^{contrib} \vdash Write(\psi^{non-free})$, and $\psi^{non-free} \vdash Read(\phi^{main})$ then $\Gamma^{contrib} \vdash Read(\phi^{main})$, by default. For querying operations this 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.

IV. 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 $\{\wedge, \rightarrow, \vee\}$, 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 $(<, \leq)$. 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 families, structurally recursive programs, pattern matching.

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 decid-

able. This typically means that terms are convertible and that a proof of `a ≡ 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:

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

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):

```
Module Profile (R: REPOSITORY) (A: ATOM R).
  Module E := Resource(R)(A).
  Include WSetsOn E.
End Profile.

Parameter typable:
  Repository.t -> P.E.t -> Prop.

Definition typable_profile
  (R: Repository.t) (P: P.t): Prop :=
  forall f, In f P -> typable R f.

Definition well_formed (P: P.t): Prop :=
  ~Empty P.

Variable Ra: Repository.t.
Variable Rb: { x | Repository.lt Ra x }.

Variable Pa: { x | typable_profile Ra x }.
Variable Pb: { x | typable_profile (`Rb) x }.
```

A SecureNDC-formula `NDProof (Pa::Pb::nil) b` expresses that a software package `b` typable in `Pb` is accessible from a profile `Pa` typable by repository `Ra`. Figure 1 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 `f1, f2` from distinct profiles `Pa,Pb` typed from different repositories, respectively `Ra,Rb`; by each of `nd_and_elim_l` and `nd_and_elim_r`, each component `f1` or `f2` of a modular package can be obtained from the combined repositories. Each of `nd_or_intro_l` and `nd_or_intro_r` allows to access a package `f1` (respectively `f2`) within an extended profile `(Pa::Pb::nil)` to form an extended package `(nd_or f1 f2)`. The corresponding elimination `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 `nd_impl_elim` establish packages dependency within a profile authorised downward in the domination 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. `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. `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:

```
Parameter typable_1_read: forall f R P P',
  typable_profile R P -> typable R f ->
  NDProof P P' (P::nil) (nd_read f).
Parameter typable_1_write: forall f R P,
  typable_profile R P -> typable R f ->
  NDProof (P::nil) (nd_write f).
Parameter typable_2_read: forall f R R' P,
  typable_profile R P -> typable R' f ->
  Repository.lt R R' ->
  NDProof (P::nil) (nd_read f).
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.

V. 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,
Inductive NDProof: list P.t -> P.E.t -> Prop :=
  | nd_atom_mess: forall b,
      well_formed (`Pa) -> typable (`Rb) b ->
      NDProof (`Pa::`Pb::nil) b
  | nd_and_intro: forall f1 f2,
      NDProof (`Pa::nil) f1 ->
      typable Ra f1 ->
      NDProof (`Pb::nil) f2 ->
      typable (`Rb) f2 ->
      NDProof (`Pa::`Pb::nil) (nd_and f1 f2)
  | nd_and_elim_l: forall f1 f2,
      NDProof (`Pa::`Pb::nil) (nd_and f1 f2) ->
      typable Ra f1 -> typable (`Rb) f2 ->
      NDProof (`Pa::`Pb::nil) f1
  | nd_and_elim_r: forall f1 f2,
      NDProof (`Pa::`Pb::nil) (nd_and f1 f2) ->
      typable Ra f1 ->
      typable (`Rb) f2 ->
      NDProof (`Pa::`Pb::nil) f2
  | nd_or_intro_l: forall f1 f2,
      NDProof (`Pa::`Pb::nil) f1 ->
      typable Ra f1 ->
      typable (`Rb) f2 ->
      NDProof (`Pa::`Pb::nil) (nd_or f1 f2)
  | nd_or_intro_r: forall f1 f2,
      NDProof (`Pa::`Pb::nil) f2 ->
      typable Ra f1 -> typable (`Rb) f2 ->
      NDProof (`Pa::`Pb::nil) (nd_or f1 f2)
  | nd_or_elim: forall f1 f2 f,
      NDProof (`Pa::`Pb::nil) (nd_or f1 f2) ->
      typable Ra f1 -> typable (`Rb) f2 ->
      NDProof (P.singleton f1::nil) f ->
      NDProof (P.singleton f2::nil) f ->
      typable Ra f ->
      NDProof (`Pa::`Pb::nil) f
  | nd_impl_intro: forall f1 f2,
      NDProof (`Pa::P.singleton f1::nil) f2 ->
      typable (`Rb) f1 ->
      typable (`Rb) f2 ->
      NDProof (`Pa::nil) (nd_impl f1 f2)
  | nd_impl_elim: forall f1 f2,
      NDProof (`Pa::nil) (nd_impl f1 f2) ->
      typable (`Rb) f1 ->
      typable (`Rb) f2 ->
      NDProof (`Pa::nil) f1 ->
      NDProof (`Pa::P.singleton f1::nil) f2
  | nd_read_intro: forall f,
      well_formed (`Pa) ->
      typable (`Rb) f ->
      NDProof (`Pa::nil) (nd_read f)
  | nd_trust_intro: forall f,
      typable (`Rb) f ->
      NDProof (`Pa::nil) (nd_read f) ->
      well_formed (P.add f (`Pa)) ->
      NDProof (`Pa::nil) (nd_trust f)
  | nd_write_intro: forall f,
      NDProof (`Pa::nil) (nd_read f) ->
      NDProof (`Pa::nil) (nd_trust f) ->
      typable (`Rb) f ->
      NDProof (`Pa::nil) (nd_write f).
```

Fig. 1. SECURENDC

```
   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 (`Pa::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 on without (NDProof).

```
(* up cut *)
Inductive NDUCProof: list P.t ->
    P.E.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 (`Pa::nil) f ->
      NDUCProof (`Pb::singleton f::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 (`Pb::`Pa::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.

VI. 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 2 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 b 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

```
Axiom TMI_atom_mess:
      forall Pa Pb b H1 H2,
      TrMinInst
      (nd_atom_mess Pa Pb b H1 H2) = 0.
Axiom TMI_and_intro:
      forall Pa Pb f1 f2
      H1 H2 T11 T12 T21 T22 n m,
      TrMinInst H1 = n ->
      TrMinInst H2 = m ->
      TrMinInst (nd_and_intro Pa Pb f1 f2
      H1 T11 T12 H2 T21 T22) = n + m.
Axiom TMI_and_elim_l:
      forall Pa Pb f1 f2 H T11 T12 T21 T22 x,
      TrMinInst H = x ->
      TrMinInst (nd_and_elim_l Pa Pb f1 f2
      H T11 T12 T21 T22) = x.
Axiom TMI_and_elim_r:
      forall Pa Pb f1 f2 H T11 T12 T21 T22 x,
      TrMinInst H = x ->
      TrMinInst (nd_and_elim_r Pa Pb
      f1 f2 H T11 T12 T21 T22) = x.
Axiom TMI_or_intro_l:
      forall Pa Pb f1 f2 H T11 T12 T21 T22 x,
      TrMinInst H = x ->
      TrMinInst (nd_or_intro_l Pa Pb
      f1 f2 H T11 T12 T21 T22) = x.
Axiom TMI_or_intro_r:
      forall Pa Pb f1 f2 H T11 T12 T21 T22 x,
      TrMinInst H = x ->
      TrMinInst (nd_or_intro_r Pa Pb
      f1 f2 H T11 T12 T21 T22) = x.
Axiom TMI_or_elim:
      forall Pa Pb f1 f2 f
      H T11 T12 T21 T22 H1 H2 T x n m,
      TrMinInst H = x -> TrMinInst H1 = n ->
      TrMinInst H2 = m ->
      TrMinInst (nd_or_elim Pa Pb f1 f2 f
      H T11 T12 T21 T22 H1 H2 T) =
      x + (max n m).
Axiom TMI_nd_impl_intro:
      forall Pa Pb f1 f2 H T11 T12 T21 T22 x,
      TrMinInst H = x ->
      TrMinInst (nd_impl_intro Pa Pb f1 f2
      H T11 T12 T21 T22) = x.
Axiom TMI_impl_elim:
      forall Pa Pb f1 f2
      H1 H2 T11 T12 T21 T22 n m,
      TrMinInst H1 = n ->
      TrMinInst H2 = m ->
      TrMinInst (nd_impl_elim Pa Pb f1 f2
      H1 T11 T12 T21 T22 H2) = min n m.
Axiom TMI_read_intro:
      forall Pa f H t,
      TrMinInst (nd_read_intro Pa f H t) = 0.
Axiom TMI_trust_intro:
      forall Pa f H Hwf x,
      TrMinInst H = x ->
      TrMinInst
      (nd_trust_intro Pa f H Hwf) = x + 1.
Axiom TMI_write_intro:
      forall Pa f H1 H2 t n m,
      TrMinInst H1 = n -> TrMinInst H2 = m ->
      TrMinInst
      (nd_write_intro Pa f H1 H2 t) = n + m.
```

Fig. 2. The Algorithm TrMinInst

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 offer a definition of the minimally trusted installation problem as follows:

Definition 6 (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 we know how to compute the number of required imports. 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 allowing (NDUCProof (Pa::Pb::nil) nd_write fb) according to the output of 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.

VII. Conclusions

We have presented an implementation in the theorem prover Coq of the typed natural deduction calculus SecureND, embedding a notion of trust as an explicit attribute of installation profiles over software packages retrievable from repositories. We use it to reformulate the Minimum Install Problem with an input space in terms of trusted packages; we resolve it with an optimization algorithm. By the structure of our typed calculus, packages reveal information about the originating repositories, hence this second parameter can also be extracted by our optimization problem. We focused on cut-elimination and we have illustrated its meaning for trust-based operations in software management systems. We have offered a solution to the mentioned Minimally Trusted Install Problem in terms of authorising the minimal number of transitive trust dependencies from external repositories. Such a solution is interpreted in terms of an algorithm to compute the minimal number of cut-rule applications. For the future we 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. A further extension can be considered in terms of an untrust function.

References

[AGA13] Nasir Ali, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Trustrace: Mining software repositories to improve the accuracy of requirement traceability links. *IEEE Trans. Software Eng.*, 39(5):725–741, 2013.

[BDS11] Sven Bugiel, Lucas Vincenco Davi, and Steffen Schulz. Scalable trust establishment with software reputation. In *Proceedings of the Sixth ACM Workshop on Scalable Trusted Computing*, STC '11, pages 15–24, New York, NY, USA, 2011. ACM.

[BPR15] J. Boender, G. Primiero, and F. Raimondi. SecureNDC - coq implementation of the SecureND calculus, March 2015. https://github.com/gprimiero/SecureNDC.

[BS04] Moritz Y. Becker and Peter Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *In IEEE Computer Security Foundations Workshop*, pages 139–154. IEEE Computer Society Press, 2004.

[CCX09] Stephen Clarke, Bruce Christianson, and Hannan Xiao. Trust*: Using local guarantees to extend the reach of trust. In Bruce Christianson, James A. Malcolm, Vashek Matyas, and Michael Roe, editors, *Security Protocols Workshop*, volume 7028 of *Lecture Notes in Computer Science*, pages 171–178. Springer, 2009.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic, Volume I.* North-Holland, 1958. Second printing 1968.

[CH88] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76(2–3):95–120, 1988.

[CH96] Bruce Christianson and William S. Harbison. Why isn't trust transitive? In T. Mark A. Lomas, editor, *Security Protocols Workshop*, volume 1189 of *Lecture Notes in Computer Science*, pages 171–176. Springer, 1996.

[CNS03] 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.

[CSW08] Peter C. Chapin, Christian Skalka, and Xiaoyang Sean Wang. Authorization in trust management: Features and foundations. *ACM Comput. Surv.*, 40(3), 2008.

[CVW$^+$11] 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.

[GKRT04] R. Guha, Ravi Kumar, Prabhakar Raghavan, and Andrew Tomkins. Propagation of trust and distrust. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 403–412, New York, NY, USA, 2004. ACM.

[GM82] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[GS00] T. Grandison and M. Sloman. A survey of trust in internet applications. *Communications Surveys Tutorials, IEEE*, 3(4):2–16, Fourth 2000.

[How80] 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.

[JP05] Audun Jøsang and Simon Pope. Semantic constraints for trust transitivity. In Sven Hartmann and Markus Stumptner, editors, *APCCM*, volume 43 of *CRPIT*, pages 59–68. Australian Computer Society, 2005.

[LM03] Ninghui Li and John C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, PADL '03, pages 58–73, London, UK, UK, 2003. Springer-Verlag.

[MBC$^+$06] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the complexity of large free and open source package-based software distributions. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*, pages 199–208. IEEE Computer Society, 2006.

[MD05] Stephen Marsh and MarkR. Dibben. Trust, untrust, distrust and mistrust an exploration of the dark(er) side. In Peter Herrmann, Valrie Issarny, and Simon Shiu, editors, *Trust Management*, volume 3477 of *Lecture Notes in Computer Science*, pages 17–33. Springer Berlin Heidelberg, 2005.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory: Lecture Notes*. Bibliopolis, Napoli, 1984.

[PR14] Giuseppe Primiero and Franco Raimondi. A typed natural deduction calculus to reason about secure trust. In Ali Miri, Urs Hengartner, Nen-Fu Huang, Audun Jøsang, and Joaquín García-Alfaro, editors, *2014 Twelfth Annual International Conference on Privacy, Security and Trust, Toronto, ON, Canada, July 23-24, 2014*, pages 379–382. IEEE, 2014.

[TSJL07] 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.

[YP11] Zheng Yan and Christian Prehofer. Autonomic trust management for a component-based software system. *IEEE Trans. Dependable Sec. Comput.*, 8(6):810–823, 2011.

[Zel13] Andreas Zeller. Can we trust software repositories? In Jrgen Mnch and Klaus Schmid, editors, *Perspectives on the Future of Software Engineering*, pages 209–215. Springer Berlin Heidelberg, 2013.

[ZL05] Cai-Nicolas Ziegler and Georg Lausen. Propagation models for trust and distrust in social networks. *Information Systems Frontiers*, 7(4-5):337–358, December 2005.