

Secure Sharing Between Untrusted Users in a Transparent Source/Binary Deployment Model

Eelco Dolstra
Utrecht University, P.O. Box 80089
3508 TB Utrecht, The Netherlands
eelco@cs.uu.nl

ABSTRACT

The Nix software deployment system is based on the paradigm of *transparent source/binary deployment*: distributors deploy descriptors that build components from source, while client machines can transparently optimise such source builds by downloading pre-built binaries from remote repositories. This model combines the simplicity and flexibility of source deployment with the efficiency of binary deployment. A desirable property is *sharing* of components: if multiple users install from the same source descriptors, ideally only one remotely built binary should be installed. The problem is that users must trust that remotely downloaded binaries were built from the sources they are claimed to have been built from, while users in general do not have a trust relation with each other or with the same remote repositories.

This paper presents three models that enable sharing: the *extensional model* that requires that all users on a system have the same remote trust relations, the *intensional model* that does not have this requirement but may be suboptimal in terms of space use, and the *mixed model* that merges the best properties of both. The latter two models are achieved through a novel technique of *hash rewriting* in content-addressable component stores, and were implemented in the context of the Nix system.

1. INTRODUCTION

Secure deployment of software is a difficult problem due to trust issues. For instance, how can we trust that binaries that we have downloaded and installed from some remote server are not malicious? Such issues are exacerbated in multi-user environments. If we allow users to install software components on their own, under what circumstances is it safe for other users to use those very same components?

Consider a typical Unix system where components are installed in (say) the directory `/usr`, or similarly `C:\Program Files` on a Windows system. Only the site administrator has the appropriate permissions to do so. This is a highly *monolithic* security model: the administrator globally makes component selections for all users, along with a determination of whether these components can be trusted.

Alternatively, individual users can install components in some location where they do have write permission, e.g., `/home/alice`. But in this case there is no sharing: if another user installs the same software (say, to `/home/bob`) duplication will result. This is bad because it increases resource consumption, e.g., disk space, disk cache, and network bandwidth.

Also, user-installed software is typically outside of the control of the deployment system (e.g., RPM in the Linux world [8]). This is an important restriction because deployment tools are supposed to track dependencies between installed components. For instance, if Alice locally installs a component *X* that depends on some globally installed component *Y*, then *Y* should not be removed while *X* is still present.

A better and more flexible model is one where both administrators and users can just install software from some trusted source — such as an operating system distributor, a site-local “channel”, or some third-party software vendor — and all such components will end up in a shared component pool, such that components will be shared between users *if and only if* they are “equal”, under some notion of component equality.

That is, users should be logically distinct with regard to the deployment system, i.e., they have the ability to install software themselves; but if a user installs a component previously installed by another user, there will be automatic sharing.

In this paper we explore such a model in the context of the *Nix deployment system* [6, 5, 4]. We have previously argued that Nix has many advantages for software deployment:

- **Reliable dependencies:** we store components in isolation from each other to enable reliable detection of the dependency graph between components, which allows us to prevent missing dependencies.
- **Side-by-side deployment of versions and variants:** if two components differ in any way (in terms of their build inputs), they do not overwrite each other, and therefore the installation of a component can never interfere with the operation of previously installed components.
- **Transparent source/binary deployment.** Components are deployed through *Nix expressions* that build components from source, which is a convenient and flexible deployment model. Source builds are slow, however; but we regain the efficiency of binary deployment in a transparent way by downloading pre-built binaries from remote repositories.
- **Safe and automatic garbage collection of unused components.**
- **Separation of installation and activation:** users can have different views on the set of installed components.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

The latter point appears to make Nix ideal for multi-user environments, except that Nix until now lacked a security model. In particular, transparent source/binary deployment requires that when we download a binary from a remote server instead of building from source, we must *trust* that that binary has actually been built from the same source and has not been tampered with. However, the Nix model assumes that all binaries resulting from a source build action are interchangeable, and allows only one such binary to be present in the system at the same time. Thus, the trust relation with remote servers must be the same for all users, which is not the case in general. (Indeed, some local users themselves might not be trustable.) That is, all users with installation rights must trust each other not to install Trojan horses (malicious components masquerading as legitimate software) or other malware.

In this paper we improve the Nix system by developing a security model that allows users to safely share installed components. The contributions of this model are the following:

- A transparent source/binary model that supports secure and automatic sharing between users *if* they are installing the same binaries, or if they have a trust relation. Sharing is enabled through the use of a *content-addressable component store*, which stores each component under a file name that is a *cryptographic hash of the contents of the component*.
- We show how content-addressable stores can deal with self-referential components (components that contain their own file name) through the technique of *hash rewriting*.
- The resulting model liberates us from the monolithic security assumptions implicit in most deployment systems (e.g., RPM). At the same time, it enables sharing which is absent from non-monolithic models (e.g., Mac OS X application bundles).
- We show how we can have unconditional sharing in a purely source-based deployment model.
- These contributions are useful not only for deployment but also for *build management*: it allows developers to transparently share build results, an improvement over tools such as Make [7].

This paper is organised as follows. In Section 2 we give an overview of the previous Nix model, and give its semantics. We show in Section 3 that in that model we can at least obtain secure sharing of *locally* built components. The main contribution is in Section 4, where we describe the new content-addressable model that allows sharing between untrusted users. We extend it in Section 5 by improving sharing between trusted users. We discuss further advantages of the new model in Section 6, and related work in Section 7.

2. THE EXTENSIONAL MODEL

2.1 Overview

In this section we describe a simplified and more powerful version of the model described in [6]¹, which we here refer to as the *extensional model* for reasons explained below.

Nix obtains its main advantages — reliable dependencies, side-by-side deployment of versions and variants, and separation of

¹The main difference is the removal of explicit closure representations from the Nix store; the references graph is now maintained per path in a database.

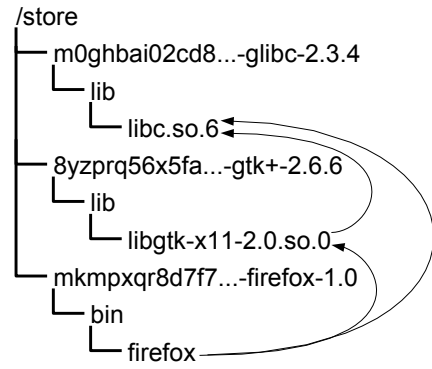


Figure 1: Nix store

component installation and activation — by storing components in isolation from each other in a *Nix store*, which is simply a directory in the file system that contains components. Figure 1 shows an instance of the Mozilla Firefox component with some of its runtime dependencies (the GUI library GTK, and the C library Glibc). Each component, which can be a single file or a whole directory tree, has a name — its *store path* — that uniquely identifies the component, e.g., `mkmpxqr8d7f7...-firefox-1.0`. The prefix is a base-32 representation of a 160-bit *cryptographic hash* [13] of all inputs involved in building the component, such as sources and dependencies. Thus, any change to the inputs yields a new component with a new store path.

The use of hashes in paths enables reliable identification of dependencies in two ways. First, it prevents undeclared build-time dependencies, for if a store path is not explicitly declared as an input to the build process, tools such as compilers and linkers will not find it (in contrast to “global” directories such as `/usr/lib`).

Second, it allows us to discover runtime dependencies by *scanning* for the hash parts of store paths inside components [6]. For instance, Unix executables contain a dynamic library search path (the *RPATH* [16]) by which the dynamic linker can find libraries. E.g., the Firefox executable in `/store/mkmpxqr8d7f7...-firefox-1.0/bin/firefox` contains in its *RPATH* the path `/store/8yzprq56x5fa...-gtk+-2.6.6/lib`. By scanning for the hash part `8yzprq56x5fa...` inside files of the Firefox component, we thus discover that it has a dependency on a specific instance of GTK 2.6.6. This technique is generic; Nix knows nothing about the format of Unix executables specifically.

Full information about the runtime dependency graph allows safe deployment by always deploying *closures* of paths under the dependency relation discovered by scanning. E.g., if we deploy our Firefox instance, we must also deploy the instances of GTK and Glibc (the C library) thus found.

Users can have different views on the set of installed applications through *user environments*, which are simply sets of symbolic links [14] to the programs of the components that each user has selected. Each user can have a different user environment.

Nix components are built from *Nix expressions* (described more fully in [5]), which is a simple functional language that describes how to build a component and all its dependencies. An example of a Nix expression to build Mozilla Firefox is shown in Figure 2. This expression is actually a *function* taking parameters `stdenv`, ..., `gtk`, representing dependencies. That is, Firefox has a build-time dependency on components such as `gtk`. To build a concrete component instance, the function should be called with actual arguments for these parameters. This is done in Figure 3.

```

{ stdenv, fetchurl, pkgconfig
, gtk, libIDL, ... }:

stdenv.mkDerivation {
  name = "firefox-1.0";

  builder = ./builder.sh;
  src = fetchurl {
    url = ftp://.../firefox-1.0-src.tar.bz2;
    md5 = ...;
  };

  buildInputs = [pkgconfig gtk ...];
}

```

Figure 2: Nix expression for Firefox

```

rec {
  firefox = (import ./firefox.nix) {
    inherit fetchurl stdenv
      pkgconfig gtk libIDL ...;
  };

  gtk = (import ./gtk.nix) {
    inherit fetchurl stdenv glib atk ...;
  };

  libIDL = ...;
  fetchurl = ...;
  stdenv = ...;
}

```

Figure 3: Nix expression composing Firefox, GTK, etc.

The result of the function is a call to `mkDerivation`, which performs a build action, or *derivation*. A build is performed by recursively building all subderivations (the dependencies passed in), then calling the script indicated by the `builder` attribute to do the actual work. The builder is typically just a simple shell script. The locations of the dependencies are passed through environment variables corresponding to the attributes of the derivation. For instance, the environment variable `gtk` will hold the path of the GTK instance (e.g. `/store/8yzprq56x5fa...-gtk+-2.6.6`). Also, the special variable `out` contains the store path where the builder is to install its output (e.g., `/store/mkmpxqr8d7f7...-firefox-1.0`).

Nix expressions are not built directly; rather, they are translated to the more primitive language of *store derivations*, which encode single component build actions. Store derivations are placed in the Nix store, and as such have a store path too. The advantage of this two-level build process is that the paths of store derivations give us a way to uniquely identify objects of source deployment, just as paths of binary components allow us to uniquely identify objects of binary deployment.

A derivation specifies the *output path* that it builds, the paths of its input derivations (build time dependencies), the paths of its immediate sources (which are copied to the Nix store by the translation process), the path of the *builder*, and the shell environment and command-line arguments to be passed to the builder. For example, the store derivation resulting from translating the `firefox` variable in Figure 2 resides in a store path `/store/rax19fjg9691-firefox.drv`

containing:

```

{ output = "/store/mkmpxqr8d7f7...-firefox-1.0"
, inputDrvs = {
  "/store/0qcsmjdk9xmd...-stdenv.drv",
  "/store/27fv8qak30hk...-gtk.drv", ... }
, inputSrcs = { "/store/m6brsfmpa31...-builder.sh" }
, builder = "/store/m6brsfmpa31...-builder.sh"
, envVars = {
  ("out", "/store/mkmpxqr8d7f7...-firefox-1.0"),
  ("stdenv", "/store/vq5r7adr687p...-stdenv"),
  ("gtk", "/store/8yzprq56x5fa...-gtk+-2.6.6"),
  ... }
}

```

Note that the environment variables refer to paths produced by the input derivations. For instance, the path `/store/8yzprq56x5fa...-gtk+-2.6.6` is the output of the derivation `/store/27fv8qak30hk...-gtk.drv`.

How are the hash parts of store paths computed? For sources and store derivations, they are the hash of the content of the file. That is, when copying a source file at path p to the Nix store, the hash is

`hash("src:" + contents(p))`

where the function `contents(p)` computes a canonical serialisation of the file system contents at path p , `hash(s)` returns a base-32 representation of a 160-bit cryptographic hash of string s , and `+` denotes string concatenation. In effect, those parts of the Nix store representing copied sources can be said to be *content-addressable*: if we know the content of a file, we also know its path. That is, there is a unique correspondence (disregarding hash collisions, which are infeasible to produce) between store paths and store contents.

On the other hand, for output paths (the results of derivations), we do not know the contents of a component until after it has been built — but we have to assign it a path *before* it is built! This is because for instance Unix components typically hard-code references to their own installation path into executables, libraries, and other files. Therefore we compute the hash part of the output of a derivation d as a hash of the input derivations:

`hash("out:" + show(d'))`

where d' equals d with output set to the empty string, and `show(d)` yields a string representation of the derivation d . The prefixes "src:" and "out:" are used to prevent sources and output paths from accidentally hashing to the same path.

2.2 Transparent Source/Binary Deployment

The model as described above is a *source deployment model*: component distributors deploy to clients Nix expressions that describe how to build the components from sources by running build actions. That is, Nix expressions essentially form a high-level Makefile for components. This is the model used by source-based deployment systems such as the FreeBSD Ports collection and Gentoo Linux [1]. Such a model is convenient and flexible for developers and component distributors, because it obviates the need to explicitly make binary packages, and because Nix expressions can express component variability, allowing customisation of components. On the other hand, source deployment is bad for end-users, because it is slow: installing Firefox from Figure 3 entails building not only Firefox but also all its dependencies, e.g., the C compiler and library, GUI libraries, and so on.

For this reason Nix has the notion of *transparent source/binary deployment* through its *substitute* mechanism. Distributors can pre-build specific instances of Nix expressions and make them centrally

available, e.g., on a webserver. Clients can then *register* the fact that such pre-built components are available, that is, that if we subsequently wish to build a store path p , we can do so by downloading a binary from URL u instead of building. Thus, source deployment transparently optimises into binary deployment. Moreover, if the user locally modifies Nix expressions or sources (say, to optimise for a specific environment), this might cause the output paths of the derivations to change, in which case binary deployment automatically falls back to source deployment.

2.3 Semantics

Here we formalise some aspects of the semantics of the extensional model, as a basis for the discussion of the intensional model in Section 4.

Nix maintains some meta-information about store paths in a number of database tables. The set $\text{valid} : \{\text{Path}\}$ (where Path in the universe of store paths) lists the paths that have been successfully built, added as a source, or obtained through a substitute. It does not include paths that are currently being built, or that have been left over from failed operations. (We write $\text{valid}[p]$ to denote that a path is valid.)

The mapping $\text{references} : \text{Path} \rightarrow \{\text{Path}\}$ maintains the dependency graph, i.e., the set of store paths referenced by the contents of each store path as discovered by scanning for hashes. At any time Nix maintains the *closure invariant*:

$$\forall p \in \text{Path} : \text{valid}[p] \rightarrow \forall p' \in \text{references}[p] : \text{valid}[p'] \quad (1)$$

This means that the set of valid paths is closed under the *references* relation. The *closure* of a path p is the set of all paths that might be accessed due to the execution of the component p :

$$\text{closure}(p) = \{p\} \cup \bigcup_{p' \in \text{references}[p]} \text{closure}(p')$$

The mapping $\text{substitutes} : \text{Path} \rightarrow (\text{Path} \times [\text{String}])$ stores the substitutes that have been registered by users. The right-hand side is the name of a program with its command-line arguments that should be executed to obtain the contents for the path denoted in the left-hand side, e.g., ("download-url.sh", ["http://...", ...]).

The following invariant states that references must be known for valid paths, as well as for paths that are invalid but have substitutes.

$$\forall p \in \text{Path} : (\text{valid}[p] \vee \text{substitutes}[p] \neq \epsilon) \rightarrow \text{references}[p] \neq \epsilon \quad (2)$$

(The special value ϵ indicates that no entry occurs for a value in the given mapping.) It is necessary to know the references of substitutable paths in order to maintain the closure invariant on deployment. That is, prior to installing a path through a substitute, we must first build or download its references. E.g., from Figure 1 it follows that prior to downloading Firefox we must download Glibc and GTK (in that order).

Figure 4 shows the build algorithm for derivations in pseudo-code. The operator \leftarrow denotes assignment, and $x \stackrel{\cup}{\leftarrow} y$ is shorthand for $x \leftarrow x \cup y$. *Locking* of output paths to prevent simultaneous builds of a derivation is omitted in this paper (though not in our implementation). Given a store derivation d , the algorithm builds the output by running the builder, but only if the output was not already valid *and* could not be made valid through a substitute. If we do build, we subsequently scan for references to input paths (that is, the union of closures of input sources and of output paths of input derivations), and set the path to valid. Note that the algorithm maintains the invariants at all times by wrapping database operations in transactions.

```

build(d) :
  if substitute(d.output) : return
  // Recursively build the inputs, then d itself.
  inputs  $\leftarrow \emptyset$ 
  for each p  $\in$  d.inputsSrcs:
    assert(substitute(p))
    inputs  $\stackrel{\cup}{\leftarrow}$  closure(p)
  for each d'  $\in$  d.inputsDrvs:
    build(d')
    inputs  $\stackrel{\cup}{\leftarrow}$  closure(d'.output)
  Run d.builder in an environment d.envVars
  and with arguments d.args
  // Assuming the build succeeded:
  In a database transaction:
    references[d.output]  $\leftarrow$ 
      the subset of inputs referenced in d.output
      found by scanning for the hash parts of inputs
    valid  $\stackrel{\cup}{\leftarrow}$  {d.output}

Bool substitute(p) :
  if valid[p] : return true
  if substitutes[p] =  $\epsilon$  : return false
  for each p'  $\in$  references[p]:
    if  $\neg$  substitute(p') : return false
  for each (p', args)  $\in$  substitutes[p]:
    if execution of program p with arguments args succeeds:
      valid  $\stackrel{\cup}{\leftarrow}$  {p}
      return true
  return false

```

Figure 4: Build algorithm in the extensional model

2.4 Extensionality

So why do we call this model *extensional*? The reason is that we make an assumption of *extensional equality*. Build operations in general are not *pure*: the contents that a builder stores in the output path can depend on impure factors such as the system time. For instance, linkers often store a timestamp inside the binary contents of libraries. Thus, each build of a derivation could produce a subtly different output.

However, the *model* is that such impure influences, if they exist, do not matter in any significant way. For instance, timestamps stored in files generally do not affect their operation. Hence extensional equality: two mathematical objects (such as software components) are considered extensionally equal if they behave the same way for any operation on them. That is, we do not care about their internal structure.

Thus, while any derivation can have a potentially infinite set of output path contents that can be produced by an execution of its builder, we view the elements of that set as interchangeable.

But this is a model — an assumption about builders. For instance, if a builder yields a completely different component when invoked at a certain time of day, it is beyond the scope of the model. In reality, also, we can *always* observe inequality by observing the internal encoding of the component, since that is a permissible operation. But the model is that any such observation does not take place or does not have an observable effect.

2.5 Sharing

Sharing a Nix store in the extensional model is only possible if all users of the Nix store trust each other. For instance, suppose

that Alice has obtained a Nix expression E from a trusted source, and pulls substitutes from machine X , where X is a malicious machine that provides Trojaned binaries for the output paths of the derivation produced by E . This may cause Alice's account to be compromised. If subsequently Bob installs the same expression E , but pulls from trusted machine Y , he will still obtain the Trojaned binary pulled by Alice. This is because both binaries occupy the same location in the file system, and Nix will not install another substitute if the output path is already valid.

The problem here is that machine X claims that its substitute is an output of some derivation d , but it isn't. However, since we have no way to verify such a claim, we cannot discover this fact. We have to *trust* such a claim, and hence we must have a trust relation with machine X .

3. LOCAL SHARING

In Section 2.5 we saw that sharing between machines is only possible in the extensional model if all users have the same remote trust relations. For locally-built derivations on the other hand (i.e., when not using substitutes), we *can* allow mutually untrusted users. The trick is in preventing a user from influencing the build for some derivation d in such a way that the result is no longer a legitimate output of d .

For instance, if Alice has direct write access to the Nix store, she can start a build of derivation d , then overwrite the output path with a Trojan horse. Similarly, even if builds are done through a server process that executes builds on behalf of users but running under a different user ID (uid), Alice can interfere with the build of d by starting a build of a specially crafted derivation d' , the builder of which writes a Trojan horse to the output path of d .

We can prevent this as follows. First, users no longer have direct write access to the Nix store. All builds are performed by a Nix server process on behalf of users. The server runs builders under user IDs that are distinct from those of ordinary user or system processes (e.g., `nix-build-{1, 2, ...}`). Also, no two concurrent builds can have the same uid. This prevents one builder from interfering with the output of another, as illustrated above. Thus, the server maintains a “pool” of free and in-use uids that can be used for building.

When a build finishes, prior to marking the output path as *valid*, we do the following:

- Ensure that there are no processes left running under the uid selected for the builder. On Linux this can be done by performing a `kill(-1, SIGKILL)` operation while running as that uid. The POSIX standard however does not specify this behaviour. In general, killing all processes running under a certain uid is tricky as it is fraught with race conditions.
- Change ownership of the output to the global Nix user.
- Remove write permission and any `setuid` or `setgid` bits (which are special permission bits on files that cause them to be executed with the rights of a different user or group — a possible security risk).

Note that the latter two steps have a subtle race condition. For instance, if we change ownership first, we have the risk of inadvertently creating a `setuid` binary owned by the global Nix user². If however we remove write and `setuid` permission first, a left-over

²Exactly to prevent this race, Linux removes `setuid` permission when changing ownership. However, this is not standard behaviour.

process spawned by the builder could restore those permissions *before* the ownership is changed. This is why the first step is important. Also, on Unix, if a left-over process opened a file before the ownership changes, it can still write to it after the change, since permissions are only checked on opening.

In conclusion, in the extensional model we can securely do source based deployment *with sharing*. Of course, that is not enough: we also want transparent binary deployment through the substitute mechanism.

Note that none of this ensures that binary components can be trusted. What it does ensure that *if* any user builds a Nix expression, the result will be a binary built by that Nix expression without outside interference. This implies that if the Nix expression is trusted, any locally built binary produced by it is also trusted.

4. THE INTENSIONAL MODEL

As we saw in Section 3, we can have secure sharing of locally built derivation outputs, but not of remotely built outputs obtained through the substitute mechanism. All users have to trust that the contents produced on another machine purportedly from some derivation d is indeed from derivation d . As stated above, such a trust relation must be global for a Nix installation. In this section we develop a substantially more powerful model in which this is not the case. We do this by moving to a *fully content-addressable Nix store* for all store objects, including derivation outputs. As we shall see, this is not trivial due to self-referential components.

In the example in Section 2.5 Alice and Bob had different trust relations mapping different outputs onto the same store paths. This problem does not exist in a content-addressable store, where the hash component of the store path is equal to the hash of the contents of that path. In such a system content equality implies path equality.

If we have this property, then different users can have different trust relations: *for each user* we can have a *different* derivation to output path mapping. This is the *intensional model* — equality is defined by internal contents, not observable behaviour. This model is much stronger than the extensional model, since it doesn't make the simplifying but unenforceable assumption of builder purity. Rather, intensionality is an inherent property of the system.

4.1 Content-addressability

The crucial property of the intensional model is that the Nix store is now content-addressable: if we know (the hash of) the contents of a store object, we know its store path. Formally, this means that the following *hash invariant* holds:

$$\forall p \in \text{Path} : \text{valid}[p] \rightarrow \text{hashPart}(p) = \text{hash}(\text{contents}(p)), \quad (3)$$

where `hashPart(p)` returns the hash component of path p , e.g., for `/store/mg12dly8...-firefox-1.0` it returns `mg12dly8....`. This invariant says that for all valid paths, the hash part of the store path equals the hash of the contents of that store path. For instance, the store path `/store/mg12dly8...-firefox-1.0` implies that its content has cryptographic hash `mg12dly8....`

So how does content-addressability help us to achieve secure sharing in multi-user Nix stores? The answer is that users can now independently install software, i.e., build derivations. If the results of those independent builds are the same, we get sharing; if they differ due to impurity, we do not get sharing. This applies not just to local builds but more significantly to substitutes.

In the example of Section 2.5, when Alice installs a derivation for Firefox using a Trojaned substitute from a malicious machine, the result will end up in some path, say `/store/x1cpydjlqxbw...-firefox-1.0`. If Bob installs the same derivation but using a le-

gitimate substitute, the content will differ and thus the result will necessarily be in a different store path, e.g., `/store/mg12dly8...-firefox-1.0`. His user environment will include the latter Firefox instance. Thus, he is insulated from Alice’s poor remote trust relation.

4.2 Hash rewriting

The property of content-addressability is easily stated but not so easily achieved. This is because we do not know the content hash of a component until after we have built it, but we need to supply an output path to the builder beforehand so that it knows where to store the component.

We solve this problem through *hash rewriting*. The idea is that we perform a build in a store path p with a *randomly generated hash part*. Afterwards, we compute the content hash, and *rename* p to

$$p' = \text{subst}(p, \text{hashPart}(p) \rightsquigarrow \text{hash}(\text{contents}(p)))$$

(where $\text{subst}(s, r)$ is a function that applies a set of substitutions r to the string s ; substitutions are denoted as $x \rightsquigarrow y$). That is, the temporary path is changed to one that obeys the hash invariant. Note that the replacement string has exactly the same length in order not to break binary components (see Section 6.4 for a further discussion of the risks of hash rewriting).

There is a snag, however: simply renaming the temporary path doesn’t work in the case of self-references, i.e., if the binary image of a file refers to its own store path. This is quite common. For instance, the RPATH of a Unix executable mentioned in Section 2.1 frequently points to its own directory so that related library components can be found. If we rename the temporary path p to p' in such a case, the references to p will become *dangling references*, and the component probably will not work anymore.

We might be tempted to replace all occurrences of the string $\text{hashPart}(p)$ in the component with $\text{hashPart}(p')$. However, this would change the contents of the component, thereby invalidating the hash! With cryptographic hashes it is not feasible to compute a “fixed point”, i.e., a string containing the hash to which the string hashes.

We solve this problem by computing hashes *modulo self-references*. Essentially, this means that we ignore self-references when computing the hash. First, when computing the hash of $\text{contents}(p)$, we *zero out* all occurrences of the string $\text{hashPart}(p)$. That is,

$$p' = \text{subst}(p, \text{hashPart}(p) \rightsquigarrow \text{hashModulo}(\text{contents}(p), \text{hashPart}(p))) \quad (4)$$

where

$$\text{hashModulo}(s, h) = \sum_{i \in \text{find}(s, h)} (i + ":" + ":" + \text{hash}(\text{subst}(s, h \rightsquigarrow \mathbf{0})))$$

where $\text{find}(s, h)$ yields the offsets of the occurrences of the substring h in the string s , and $\mathbf{0}$ denotes a string consisting of binary 0s of the same length as h . It is necessary to encode the offsets of the occurrences of h into the hash to prevent hash collisions for strings that are equal except for having either h or 0-strings at the same location. The colons simply act as disambiguators, separating the offsets and the contents.

Second, we copy p to p' , rewriting all occurrences of $\text{hashPart}(p)$ in the contents of p with $\text{hashPart}(p')$:

$$\begin{aligned} \text{contents}(p') &= \text{subst}(\text{contents}(p), \\ &\quad \text{hashPart}(p) \rightsquigarrow \text{hashPart}(p')) \end{aligned}$$

Note that

$$\begin{aligned} &\text{hashModulo}(\text{contents}(p), \text{hashPart}(p)) \\ &= \text{hashModulo}(\text{contents}(p'), \text{hashPart}(p')) \end{aligned}$$

even though

$$\text{hash}(\text{contents}(p)) \neq \text{hash}(\text{contents}(p'))$$

in case of self-references. That is, the hash modulo the randomly generated hash part does not change after rewriting.

Finally, the hash invariant now reads

$$\forall p \in \text{Path} : \text{valid}(p) \rightarrow \text{hashPart}(p) = \text{hashModulo}(\text{contents}(p), \text{hashPart}(p)) \quad (5)$$

4.3 Semantics

We can now formalise the intensional model. The main difference with the extensional model is that output paths are no longer known *a priori*. But because of this, we cannot prevent re-building a derivation by checking (as was done in Figure 4) whether its output path is already valid. The same applies to checking for substitutes, which are also keyed on output paths.

Also, in the intensional model, due to impurity a single derivation can result in several distinct components residing at different store paths, if the derivation is built multiple times (e.g., by different users). That is, a derivation actually defines an *equivalence class* of store paths within the Nix store, the members of such classes all having been produced by the same derivation. Thus, we add to derivations a field `eqClass` : Path and remove the field `output`.

We represent equivalence classes as store paths. For instance, the Firefox derivation might have `d.eqClass` = `"/store/vh8a8r83syd5...-firefox-1.0"`. These paths are “virtual”: they are never built. The reason for using store path for equivalence classes is that it gives us an easy way to refer to the output of a derivation from other derivations. For instance, the `envVars` field of a derivation that depends on Firefox must in some way refer to the path of the Firefox component, even though this path is not known in advance anymore. When we build a derivation d depending on derivation d' , we simply rewrite in $d.\text{envVars}$ all occurrences of $\text{hashPart}(d'.\text{eqClass})$ to a trusted member of the equivalence class denoted by $d'.\text{eqClass}$. Equivalence classes are computed in exactly the same way as the `output` field was computed in the extensional model: by hashing the derivation with its `eqClass` field set to the empty string.

Since we must remember for each derivation what output paths were produced by it and who built or substituted them, we define a database mapping `eqClassMembers` : Path \rightarrow {UserId \times Path}, meaning that an equivalence class maps to a set of store paths along with the name of the user that built or substituted the store path. A store path can occur multiple times for different users. We also maintain a reverse mapping `eqClasses` : Path \rightarrow {Path} that maps a store path to the set of equivalence classes of which it is a member. Note that a path can be in multiple equivalence classes: this happens if different derivations happen to produce exactly the same output.

Similarly, we also extend the `substitutes` mapping with a `UserId` for each substitute to indicate which user registered it.

The set of *trusted paths* in the equivalence class of a derivation output is simply the set of valid or substitutable paths for some user:

$$\begin{aligned} \text{trustedPaths}(\text{eqClass}, \text{user}) &= \\ &\{p \in \text{Path} \mid (\text{user}, p) \in \text{eqClassMembers}[\text{eqClass}]\} \end{aligned}$$

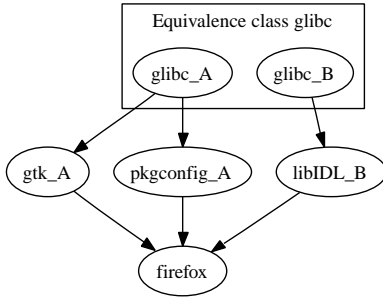


Figure 5: An equivalence class collision

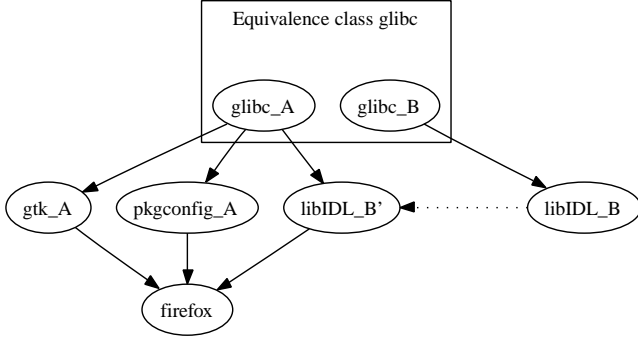


Figure 6: Resolution of the equivalence class collision

4.3.1 Equivalence class collisions

The fact that a derivation can resolve to any number of output paths due to impurity, leads to the problem that we might end up with a closure that contains more than one element from the output equivalence class of a derivation.

Figure 5 shows an example of this. Suppose that Alice has locally built `gtk` and `pkgconfig` (which both depend on `glibc`). She has also registered Bob’s remotely built `libIDL` as a substitute (which also depends on `glibc`). However, though Bob’s `glibc` was built from the same derivation, due to impurities the build result is different. Thus, $\text{eqClassMembers}(\text{glibc.drv}) = \{(\text{Alice}, \text{glibc}_A), (\text{Alice}, \text{glibc}_B)\}$. This is in itself not a problem. However, suppose that Alice next tries to build `firefox`, which depends on `gtk`, `pkgconfig`, and `libIDL`. We then end up with a Firefox binary that links against two `glibc`s. This might work, or it might not — depending on the exact semantics of dynamic linking. In any case, it is an observable effect — it influences whether a build succeeds and whether the build result works properly.

Thus, we need to prevent that any closure ever contains more than one path from an equivalence class. This is the *equivalence class uniqueness invariant*:

$$\begin{aligned} \forall p \in \text{Path} : \text{valid}[p] \rightarrow \forall p_1, p_2 \in \text{closure}(p) : p_1 \neq p_2 \rightarrow \\ \forall e_1 \in \text{eqClasses}[p_1] : \forall e_2 \in \text{eqClasses}[p_2] : e_1 \neq e_2 \end{aligned} \quad (6)$$

That is, two different paths in a closure cannot be in the same equivalence classes.

Thus, when we build a derivation, we have to select from the paths in the union of input closures *one from each equivalence class*. However, we must still maintain the closure invariant (equation 1). For instance, in Figure 5, we cannot simply select the set

$\{\text{glibc}_A, \text{gtk}_A, \text{pkgconfig}_A, \text{libIDL}_B\}$, since `libIDLB` depends on `glibcB` which is not in this set.

Once again, hash rewriting comes to the rescue. We can *rewrite* `libIDLB` so that it refers to `glibcA` instead of `glibcB`. That is, we compute a new path `libIDLB'` with contents

$$\text{subst}(\text{contents}(\text{libIDL}_B), \text{hashPart}(\text{glibc}_B) \rightsquigarrow \text{hashPart}(\text{glibc}_A))$$

(Of course, self-references in `libIDLB` must also be rewritten as described in Section 4.2.) This is shown in Figure 6. The dotted edge denotes a copy-with-rewriting action.

An interesting problem is *which* paths to select from each equivalence class such that the number of rewrites is minimised. For instance, if we select `glibcA`, then we have to rewrite one path (namely `libIDLB`), while if we select `glibcB`, we have to rewrite two paths (`gtkA` and `pkgconfigA`). We do not currently know whether there exists an efficient algorithm to find an optimal solution. A heuristic that works fine in practice is to do a bottom-up traversal of the equivalence class dependency graph, picking from each class the path that induces the least number of rewrites.

However, picking an optimal solution with respect to the current derivation is not particularly useful in any case, since this ignores both the state of the Nix store as a whole, and future derivations. For instance, in our example Alice might in the future install many additional components from Bob’s remote repository (e.g., because Bob is a primary distribution site). Thus, globally there are many more paths referring to `glibcB` than to `glibcA`. In this case it is better to select `glibcB` and rewrite Alice’s locally built components. Possible heuristics include:

- Select the path that has the largest *referer closure* in the Nix store. The referer relation is the inverse of the references relation. Thus, the referer closure of p is the set of all paths directly or indirectly referencing p .
- Select the path that is also trusted by the system administrator (e.g., a special user named `root`).
- Select the path that was obtained from a substitute referring to a distribution site with a special status.

Figure 7 shows the resolution algorithm. The function `resolve` accepts a set of paths closed under the references relation and possibly violating invariant 6, and yields a set of paths that does obey the invariant. The function `selectPaths` must implement some policy of selecting a path from each equivalence class, as discussed above. However, this set is not necessarily closed, so paths referring to paths outside of the set must be rewritten to refer only to paths in the set. This is done by `maybeRewrite`, which inspects the references of path p , maps them onto equivalent paths in the selected set, and recursively rewrites them. Then, if any of the references changed, p itself is rewritten. (For brevity, memoisation of `maybeRewrite` is omitted.)

The auxiliary function `copy` copies a store path after applying a set of hash rewrites to the contents. This is the only function that adds valid paths to the store in the intensional model.

4.3.2 Build algorithm

Figure 8 shows the build algorithm for the intensional model. As in Section 3, we assume that all operations on the Nix store are done by a privileged user on behalf of the actual users, who do not have write access themselves. The crucial differences with the build algorithm for the extensional model (Figure 4) are as follows. We call `resolve` to fix equivalence class collisions. We rewrite occurrences of equivalence classes in the environment variables and

```

resolve(paths) :
  // For each path determine its equivalence class.
  for each p ∈ paths : for each e ∈ eqClasses[p] :
    conflicts[e] ← ∪ {p}
  // For all e, select one path from conflict[e].
  selected ← selectPaths(conflicts)
  paths' ← ∅
  for each p ∈ selected :
    paths' ← ∪ {maybeRewrite(p, selected)}
  return paths'

Path maybeRewrite(p, selected) :
  newRefs ← ∅
  rewrites ← ∅
  for each p' ∈ references[p] :
    p'' ← the path in selected with same equivalence class as p'
    p''' ← maybeRewrite(p'', selected)
    newRefs ← ∪ {p'''}
    rewrites ← ∪ (hashPart(p') ~ hashPart(p'''))
  if newRefs = references[p] : return p
  return copy(p, newRefs, rewrites)

Path copy(p, refs, rewrites) :
  c ← subst(contents(p), rewrites)
  // Compute the new path according to Eq. 4.
  h ← hashModulo(c, hashPart(p))
  p' ← subst(p, hashPart(p) ~ h)
  if ¬ valid[p'] :
    c' ← subst(c, hashPart(p) ~ h)
    store contents c' at path p'
    valid ← ∪ {p'}
    references[p'] ← refs
  return p'

```

Figure 7: Collision resolution algorithm

builder location to the actual paths selected by `trustedPaths` and `resolve`. For the output path, we construct a temporary path equal to $d.eqClass$ but with a random hash part. After the build, we copy and rewrite this temporary path to its final, content-addressable location in the Nix store. The temporary path can then be garbage collected.

The omitted function `substitute(p)` is as in the extensional model, except that the substitute program produces a temporary path which we then `copy` to its final location. Also, it only tries substitutes registered by the current user.

5. THE MIXED MODEL

The intensional model described in the previous section gives us a Nix store that can be shared by mutually untrusted users, or users who have different remote trust relations. Due to content-addressability, we get sharing between multiple builds of a derivation if each build produced exactly the same binary result, that is, if there is no impurity in the build. If there is impurity, then each build result will end up under a different store path.

Between untrusted users, this is exactly what we want. For instance, if Alice obtains substitutes from a malicious machine, it does not affect Bob. Note that Alice and Bob do automatically get sharing if they happen to get their substitutes from the same remote machine.

However, we would like to re-enable sharing in common scenar-

```

build(d) :
  // Note: curUser is the invoking user.
  trusted ← trustedPaths(d.eqClass, curUser)
  for each p ∈ trusted :
    if substitute(p) : return

  // Gather all trusted input closures, then resolve.
  inputs ← ∅
  for each p ∈ d.inputsSrcs :
    assert(substitute(p))
    inputs ← ∪ closure(p)
  for each d' ∈ d.inputsDrvs :
    build(d')
    for each p ∈ trustedPaths(d'.eqClass, curUser) :
      inputs ← ∪ closure(p)
  inputs ← resolve(inputs)

  // Rewrite equivalence classes to real paths.
  mapping ← ∅
  for each p ∈ inputs : for each e ∈ eqClasses[p] :
    mapping ← ∪ (hashPart(e) ~ hashPart(p))
  Apply substitution mapping to d.envVars and d.builder

  // Build in a temporary path.
  output ← subst(d.eqClass,
    hashPart(d.eqClass) ~ randomHash())
  d.envVars["out"] ← output
  Run d.builder in an environment d.envVars
  and with arguments d.args
  refs ← the subset of inputs referenced in output
  output' ← copy(output, refs, ∅)
  eqClassMembers[d.eqClass] ← ∪ {(curUser, output')}
  eqClasses[output'] ← ∪ {d.eqClass}

```

Figure 8: Build algorithm in the intensional model

ios. For instance, users generally trust components installed by the administrator. Thus, if Alice is an administrator, then Bob should be able to use the output paths already installed by Alice. In general, users should be able to specify *trust relations* between each other.

We can achieve this through a simple extension of the intensional model called the *mixed model*. For each user, we maintain a mapping `trustedUsers` : `UserId` → `{UserId}` that specifies for each user a set of trusted users. E.g., `trustedUsers["bob"] = {"alice", "bob"}`. (The mapping should be reflexive, i.e., $u \in \text{trustedUsers}[u]$.) We then augment the function `trustedPaths`:

```

trustedPaths(eqClass, user) =
  {p ∈ Path | ∃ u ∈ trustedUsers(user) :
    (u, p) ∈ eqClassMembers(eqClass)}

```

Otherwise, this is exactly the intensional model. Of course, sharing between users increases the possibility of equivalence class collisions, but that is handled through the resolution algorithm in Section 4.3.1. The name “mixed model” does not imply that we back away from intensionality — the store is still fully content-addressed. We just gain back the ability to have sharing between users. The crucial difference with the extensional model is that sharing is now selective and fine-grained.

6. DISCUSSION

In this section we briefly discuss two other significant advantages of the intensional model over the extensional model — support for *multiple outputs* and for more efficient *build management*; as well as the security implications of content-addressability through hashing, and our experiences with the implementation thereof.

6.1 Multiple outputs

Up till now, we have only considered derivations with one output path. It is very useful to allow multiple outputs to obtain finer-grained deployment. For instance, the output of the Firefox derivation might be separated into parts *program* and *library*, where the latter contains Firefox’s page rendering libraries. This allows other components to depend on the rendering libraries without requiring the installation of the main Firefox program code.

In the extensional model this however creates an unsolvable problem: suppose for instance in the scenario above that we have performed a source build of an application that depends on the Firefox rendering libraries, and we then run the Nix garbage collector that gets rid of unreferenced components. The *program* output will be deleted; only *library* is retained since it is reachable from an installed application. Now suppose that we later require the *program* output. We cannot build the derivation again, since the *library* output is in the way. And we cannot delete *library*, because that would violate the closure invariant.

In the intensional model this is not a problem: we just perform the build again, producing outputs under randomly named temporary store paths, then delete the temporary outputs we already had, and copy to content-addressed paths the ones we didn’t have.

The outputs of a derivation can refer to each other (e.g., *program* referring to *library*), but we do not allow mutually recursive references since our hash rewriting scheme does not support it (it only handles self-references). This is not an onerous restriction, since mutual recursion between output paths requires them to be deployed and garbage collected as a unit, negating the whole point of having them as separate outputs.

6.2 Build management

Nix is typically used for large-grained building, i.e., where a Nix builder builds a full component. There is nothing preventing us from using Nix as an alternative to build tools such as Make [7], however. This happens when the build actions are small steps, such as compiling a single source file, linking, etc. Thanks to Nix’s isolation properties, missing dependencies in Makefiles are always detected.

A common efficiency problem in incremental building is that a small change to some file causes many rebuilds. For instance, a change to a C header file (say, `config.h`) forces us to rebuild all source files that include it. This is the case even for source files that include the header but are unaffected by the change.

In the intensional model, redundant rebuilds can be prevented in this case by making the C preprocessor invocation an explicit build step. For source files that are unaffected by the header change, the preprocessed output will remain unchanged. We can optimise the build algorithm as follows. If a derivation d_1 differs only from a derivation d_2 in subderivations that yielded the same output, we can just copy the output of d_2 to d_1 . Thus we get more caching than with Make-like tools.

6.3 Cryptographic hashing

A content-addressable system depends for its correctness on the assumption that collisions of the hash function being used do not occur in practice. That is, it should be computationally infeasible

to produce two inputs that hash to the same value. This is the basic goal of cryptographic hash functions [13]. Nix uses 160-bit (truncated) SHA-256 hashes, meaning that the brute-force effort required to find a collision is 2^{80} hash operations on average.

However, the possibility that cryptographic hash functions might suddenly be broken, is a threat to the long-term deployment of tools depending on their security (a concern expressed in [10]). Indeed, MD5 has recently been broken [19], and SHA-1 weakened [18].

6.4 Evaluation

The most risky part of the extensional model is the use of hash rewriting. It comes as a shock to some that this even works, i.e., doesn’t produce broken binaries. In [6], we even wrote that “patching files [by rewriting hashes] is unlikely to work in general, e.g., due to internal checksums on files being invalidated in the process.” It turns out that this assessment was too pessimistic. Whether the technique is practical is an empirical question. We have applied hash rewriting to a set of applications from the Nix Packages collection [3] consisting of 86 components, and verified that the resulting applications were functional. These applications include C/Unix programs such as Firefox, as well as Java and C# programs such as MonoDevelop. We encountered no problems.

Hash rewriting necessarily fails in case of *pointer hiding* [6], i.e., when references to components are stored in such a way that they are not detected as such (e.g., in compressed files). However, this also causes dependency scanning to fail, and those cases are very rare (in fact, to date we have not encountered them at all).

7. RELATED WORK

Nix’s transparent source/binary model is a unique feature for a deployment system. Relative to binary-only or source-only deployment models, it adds the complication that we do not just need to authenticate binaries but also the fact that they are a bona fide result of certain sources. However, caching and sharing between users of build results is a feature of some SCM systems such as Vesta [11].

As claimed in the introduction, deployment systems tend to have monolithic trust models. For instance, typical Unix package management systems such as RPM [8], Debian APT, or Gentoo Linux [1], allow installation of software by the administrator only; software installed by individual users is not managed by those tools. On the other hand, Mac OS X application bundles may be installed and moved around by any user, but the system does not track dependencies in any way.

Security aspects of deployment have typically focused on ensuring integrity of components in transit (e.g., by using signatures), and on assessing or constraining the impact of a component on the system (e.g., [17]). We have not addressed the issue of ensuring that remote substitutes have not been tampered with (e.g., by a man-in-the-middle). Obviously, such problems can be solved by cryptographically signing substitutes, or rather, the *manifests* (lists of substitutes available on a server), since the fact that substitutes themselves have not been tampered with is easily verified by comparing their cryptographic hashes to their names.

Microsoft’s .NET has a Global Assembly Cache that permits sharing of components [15]. This is however not intended for storage of components private to an application. Thus, if multiple users install an application having such private components, duplication can occur. Also, .NET has a purely binary deployment model, thus bypassing source/binary correspondence trust issues.

In [9] a scenario is described in which components impersonate other components. This would not be possible in a content-addressable file system with static component composition (e.g., Unix dynamic libraries with RPATHs pointing to the full paths of

components to link against, as happens in the Nix Packages collection).

Content-addressability is a common property of the distributed hash schemes used in peer-to-peer file-sharing and caching applications (e.g., Pastry [12]). It is also used in the storage layer of version management tools such as Monotone [2].

8. CONCLUSION

The Nix deployment system has many nice features, such as reliable dependencies, side-by-side deployment of different versions or variants of components, the ability to atomically upgrade or roll back, transparent source/binary deployment, and the ability for different users in a system to independently maintain sets of activated applications. The latter is ideal for multi-user environments, except that Nix's previous *extensional* model required mutual trust between all users. The *intensional* model described in this paper lifts that requirement, while the *mixed* model recaptures the sharing of the extensional model using fine-grained trust relations. At the same time, it allows Nix to support derivations with multiple outputs, yielding more efficient deployment, and to support more efficient low-level build management.

In [6] the techniques underlying the Nix system were motivated by analogy to techniques used in programming language implementation. For instance, scanning for hash references in files to determine possible runtime dependencies is analogous to the way conservative garbage collectors find pointers. This paper improves on that result by adding hash rewriting — the analogue of pointer rewriting in copying garbage collectors — to the repertoire of operations in the Nix deployment system.

Nix can be downloaded from [3].

9. REFERENCES

- [1] Gentoo Linux. <http://www.gentoo.org/>, 2005.
- [2] Monotone. <http://www.venge.net/monotone/>, 2005.
- [3] Nix deployment system. <http://www.cs.uu.nl/wiki/Trace/Nix>, 2005.
- [4] E. Dolstra. Efficient upgrading in a purely functional component deployment model. In *Eighth International SIGSOFT Symposium on Component-based Software Engineering (CBSE 2005)*, St. Louis, Missouri, USA, May 2005.
- [5] E. Dolstra, M. de Jonge, and E. Visser. Nix: A safe and policy-free system for software deployment. In L. Damon, editor, *18th Large Installation System Administration Conference (LISA '04)*, pages 79–92, Atlanta, Georgia, USA, November 2004. USENIX.
- [6] E. Dolstra, E. Visser, and M. de Jonge. Imposing a memory management discipline on software deployment. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 583–592. IEEE Computer Society, May 2004.
- [7] S. I. Feldman. Make—a program for maintaining computer programs. *Software—Practice and Experience*, 9(4):255–65, 1979.
- [8] E. Foster-Johnson. *Red Hat RPM Guide*. John Wiley and Sons, 2003.
- [9] M. Grechanik and D. Perry. Secure deployment of components. In W. Emmerich and A. L. Wolf, editors, *2nd International Working Conference on Component Deployment (CD 2004)*, volume 3083 of *Lecture Notes in Computer Science (LNCS)*. Springer, May 2004.
- [10] V. Henson. An analysis of compare-by-hash. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, May 2003.
- [11] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 311–320. ACM Press, 2000.
- [12] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Nov. 2001.
- [13] B. Schneier. *Applied Cryptography*. John Wiley and Sons, second edition, 1996.
- [14] W. R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1993.
- [15] C. Szyperski. *Component Software — Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, second edition, 2002.
- [16] TIS Committee. Tool Interface Specification (TIS) Executable and Linking Format (ELF) Specification, Version 1.2. <http://www.x86.org/ftp/manuals/tools/elf.pdf>, May 1995.
- [17] V. N. Venkatakrisnan, R. Sekar, T. Kamat, S. Tsipa, and Z. Liang. An approach for secure software installation. In *Proceedings of the 16th Systems Administration Conference (LISA '02)*, pages 219–226. USENIX Association, Nov. 2002.
- [18] X. Wang, Y. Yin, and H. Yu. Collision search attacks on SHA1. <http://theory.csail.mit.edu/~yiqun/shanote.pdf>, 2005.
- [19] X. Wang and H. Yu. How to break MD5 and other hash functions. In *Eurocrypt 2005*, May 2005.