Stork: Secure Package Management for VM Environments

by

Justin Angelo Cappos

A Dissertation Submitted to the Faculty of the

Department of Computer Science

In Partial Fulfillment of the Requirements

For the Degree of

Doctor of Philosophy

In the Graduate College

The University of Arizona

2 0 0 8

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Justin Angelo Cappos
entitled Stork: Secure Package Management for VM Environments
and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

_____ Date: April 14, 2008
John. H. Hartman

_____ Date: April 14, 2008
Chris Gniady

_____ Date: April 14, 2008
Larry L. Peterson

_____ Date: April 14, 2008
Beichuan Zhang

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

_____ Date: April 14, 2008
Dissertation Director: John. H. Hartman

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: __JUSTIN ANGELO CAPPOS_____

ACKNOWLEDGEMENTS

First, I would like to thank the undergraduates who assisted with the development of Stork including Matt Borgard, Mario Gonzalez, Petr Moravsky, Jude Nelson, Duy Nyugen, Peter Peterson, Collin Reynolds, Kelland Thomas, Mike White, and Byung Suk Yang. A special mention goes to Scott Baker, Jeffry Johnston, Jason Hardies, Jeremy Plichta, and Justin Samuel for their excellent contributions.

I would also like to thank all of the administrators who use Stork. Your patience, comments, and bug reports have made Stork popular and robust.

I am indebted to the creators of the services Stork uses including Vivek Pai, KyoungSoo Park, Sean Rhea, Ryan Huebsch, and Robert Adams for their efforts in answering countless questions. I would also like to thank Steve Muir, Faiyaz Amhed, and Marc Fiuczynski at PlanetLab Central for their efforts throughout the development of Stork.

Of course, none of this would be possible without the guidance of my advisor, John Hartman. His direction and advice was instrumental to shaping Stork.

I would like to thank my family and friends for their assistance. I appreciate the support my parents provided me throughout my graduate studies. I would also like to thank Wenjun Hu for her understanding and encouragement.

Table of Contents

TABLE OF CONTENTS—*Continued*

TABLE OF CONTENTS—*Continued*

TABLE OF CONTENTS—*Continued*

TABLE OF CONTENTS—*Continued*

Table of Contents—*Continued*

TABLE OF CONTENTS—*Continued*

Table of Contents—*Continued*

List of Figures

List of Figures—*Continued*

List of Figures—*Continued*

List of Tables

List of Tables—*Continued*

List of Tables—*Continued*

Abstract

Package managers are a common tool for installing, removing, and updating software on modern computer systems. Unfortunately existing package managers have two major problems. First, inadequate security leads to vulnerability to attack. There are nine feasible attacks against modern package managers, many of which are enabled by flaws in the underlying security architecture. Second, in Virtual Machine (VM) environments such as Xen, VMWare, and VServers, different VMs on the same physical machine are treated as separate systems by package managers leading to redundant package downloads and installations.

This dissertation focuses on the design, development, and evaluation of a package manager called Stork that does not have these problems. Stork provides a security architecture that prevents the attacks other package managers are vulnerable to. Stork also is efficient in VM environments and reduces redundant package management actions. Stork is a real system that has been in use for four years and has managed half a million VM instantiations.

Chapter 1

Introduction

There are many ways to manage software on modern computer systems. One of the most popular is the use of a *package manager* [5, 6, 7, 63, 72, 76, 81, 90, 92]. A package manager is a privileged program that manages the installation, update, and removal of software on the system. Software that is meant to be managed by a package manager is placed into an archive called a *package*. In addition to containing software, packages also have *package metadata* that provides information about the package, including the other packages on which this package depends to function (called *dependencies*). Packages are made available via a server called a *package repository*. Repositories are conceptually similar to web servers, except their primary purpose is to serve packages instead of web pages.

Most package managers download files from the repository, resolve package dependencies, and verify cryptographic signatures. *Dependency resolution* takes a requested package and returns a list of packages that must be installed to have all packages on the system have all of their dependent packages installed.

Cryptographic signatures are verified on a package or package metadata to detect tampering by an attacker.

However, package managers are not a panacea. There are several aspects of package management existing package managers do not handle well, including security and VM environments. First this chapter discusses the problems security and VM environments cause for package management. Then a package manager Stork is presented which solves these problems. Following that the contributions of this dissertation are outlined. Next, the importance and scope of the dissertation are described. This chapter concludes with a map of the other chapters of the dissertation.

1.1 Security

The first area in which existing package managers are lacking is security. There are problems with cryptographic signatures, key revocation, delegation of trust, and interactions with the repository.

Existing package managers do not handle cryptographic signatures well. Existing package signatures require that the package is downloaded before the signature can be verified. This means that the package metadata cannot be validated using the package signature unless the package is downloaded. The only way to verify that all of the package metadata on the repository is correct is to download all of the packages. As a result, package metadata is not verified using the package signature. This means that an attacker can create its own package metadata and package signatures do not prevent this attack.

Some package managers use an alternative approach to signatures on packages. Instead the package managers have the repository administrator sign the package metadata. This means that if an attacker tampers with package metadata it will be detected because the signature on the package metadata will not match. Unfortunately this places a high degree of trust in the repository administrator. This is problematic because the repository administrator is an intermediary and is usually not the person that packaged the software. As a result, the repository

administrator is signing metadata for packages for which she has no first-hand knowledge if the package is valid or has been tampered with. Each repository usually holds many different packages from different sources. The repository administrator signs package metadata for all of these sources. If the repository administrator incorrectly signs package metadata then the repository administrator's key must be revoked. This removes trust in all package metadata the repository administrator signed regardless of the source.

Further complicating the issue, it is common for a package or package metadata to be signed and later the signer would like to revoke the signature. For example, if a group releases a package and discovers a security hole, the group would like administrators to only install the fixed version. Unfortunately, trust revocation is not handled well in existing package managers. There is no mechanism to remove the trust of individual packages, instead the key which signed the package must be revoked. Furthermore, if a key is compromised, there is no automatic mechanism for revoking the key. The revocation must be done manually or by overloading some existing mechanism (such as installation of package that as part of the install process revokes a key).

Another security problem is that trusting package signatures is typically a very coarse-grained operation in which a key is either trusted or not. In other words, a key is either trusted to sign anything or the key is not trusted. This is problematic

because administrators will use packages from different projects that are created by different developers. Because trust is a coarse-grained operation a developer from one project can sign packages from another project and the package manager will trust the package. This allows an attacker that can compromise one project's key to use this to install malicious versions of packages for any project.

In addition there are implicit trust assumptions that negatively impact security. Existing package managers implicitly trust the repository. Unfortunately, this allows an attacker to be trusted if the attacker can intercept and alter traffic to and from the repository.

1.2 Virtual Machines

Another area that is not handled well by modern package managers is virtualization. Modern virtual machine (VM) environments such as Xen [8], VMWare [83], and VServers [43, 45], can run multiple "operating system" virtual machines on a single physical machine. This isolates VMs to provide the illusion that different VMs on a single physical machine each run on their own physical machine. Unfortunately this isolation leads to inefficient use of network bandwidth, disk space, and memory by package managers.

First, VMs are treated by package managers as if the VMs are separate physical machines. This means that when multiple VMs on the same physical machine install a package, the package is downloaded multiple times. This wastes network bandwidth due to the redundant download of packages from the repository.

Second, when a package is installed in different VMs on the same physical machine, its files are stored separately on disk. This is inefficient because it consumes extra disk space. In addition, some virtual machine monitors (VMMs) are able to share memory between multiple instances of the same software. If there are separate copies of the software running then the VMM is not able to share memory which leads to increased memory usage.

1.3 Research Questions

The core contributions of this dissertation are:

- Building a real system for package management called Stork [15, 16, 51, 69, 76] that solves the above problems. Stork does dependency resolution, supports multiple transfer protocols, and has rudimentary package-based configuration management facilities. Stork provides a complete solution to the problem of package management and has clients that are VMs as well clients that are normal systems. Stork is in everyday use — the main Stork repository serves a package roughly once every 20 seconds. Stork is used daily by administrators at about two dozen institutions. The package-based configuration management utilities are used to manage VMs on hundreds of failure prone, globally distributed, physical machines. Stork has been leveraged as a building block in the construction of other services. Stork has been used for 4 years and has managed half a million VM instantiations.

- A security architecture that avoids the security flaws in existing package managers is presented. Vulnerabilities in existing package managers including key revocation, escalation of privilege attacks, lack of appropriate metadata protection, and repository trust are analyzed. Upon uncovering these vulnerabilities, commonalities between vulnerabilities is discussed.

Unfortunately, many of the security problems in existing package managers are architectural in nature so the security problems cannot be fixed with patches.

- This dissertation demonstrates the complexities and benefits of package managers sharing packages between VMs. Two architectures are implemented to demonstrate the effects of different architectures for sharing packages. One architecture is built on the principle of keeping the implementation simple. The other architecture is built to try to minimize the trusted code base. This work illustrates the trade-offs presented by these techniques between ease of implementation and security.

1.4 Importance

It is important to solve these problems in package management because the underlying factors that caused these problems are increasing. VMs are becoming more prevalent with all major desktop operating systems supporting operating system virtualization. The number and complexity of attacks on the Internet is also increasing which further underscores the importance of protecting software installation mechanisms. These trends imply that if the problems with package management are not addressed, the situation will continue to deteriorate.

1.5 Scope

Stork can be used for different types of clients including desktop systems and VM environments. Stork also can be used to manage different security environments including clients on an isolated LAN or clients on the Internet. This dissertation evaluates Stork in two environments. To understand the effectiveness of sharing packages in VM environments Stork is evaluated on PlanetLab [62]. PlanetLab is a collection of research machines that provides operating system VMs to researchers. To understand the security of Stork, this dissertation focuses on the security issues likely to face clients connected to the Internet.

1.6 Map

The remainder of this dissertation is laid out as follows:

Chapter 2 discusses Stork, introducing the major components, the architecture, and the features of Stork. Chapter 3 describes security issues in other package managers and how security was provided in Stork. Chapter 4 discusses how to efficiently share packages in VM environments. Chapter 5 concludes the paper and describes future work. The appendix contains Chapter A which defines the terms used in this dissertation.

Chapter 2

Stork

This chapter provides a description of Stork. First, some examples are provided to demonstrate Stork's feature set and show how Stork is used in practice (Section 2.1). An overview of the different components of Stork is presented in Section 2.2. Then follows a detailed description of the components that comprise Stork: the package-based configuration management tools (Section 2.3), repository (Section 2.4), client tools (Section 2.5), and nest (Section 2.6). These components are described in sufficient detail to understand their function and purpose, however the security and sharing aspects of these components are described in more detail in Chapters 3 and 4. This chapter elaborates on the examples given in Section 2.1 to demonstrate how the described functionality is provided (Section 2.7). Section 2.8 discussed related work and Section 2.9 concludes.

2.1 Examples

This section provides examples that demonstrate Stork's feature set. These examples are revisited in Section 2.7 to explain how the described functionality is provided.

These examples describe scenarios involving clients that are managed by Stork. There are two types of clients Stork manages: VMs and systems. The term *VM* is used to describe a operating system virtual machine. The term *system* is used to describe a computer that does not run VM software. The generic term *client* is used to describe either a VM or a system. The examples in this section focus on systems. How VMs perform similar actions is covered in Chapter 4.

The examples in this section can be viewed as either as separate examples of independent actions or as one big example of how Stork works end-to-end. The first examples describe an administrator that creates *package instructions* using a GUI (Section 2.1.1) or command line tools (Section 2.1.2) to control the *package actions* enacted by systems. After the administrator creates files containing the package instructions, these files are uploaded to a Stork repository. A brief description of how the repository handles uploaded files is described in Section 2.1.3 (with security related details deferred to Chapter 3). The repository notifies clients that there is new content on the repository (Section 2.1.4). The clients download updated

information from the repository to have a local and current copy of the repository's metadata (Section 2.1.5). Security concerns related to handling data retrieved from the repository are deferred to Chapter 3. The next example (Section 2.1.6) describes how a system uses the package instructions from the administrator to decide which package actions to perform. The final four sections describe how different package actions are performed. These sections provide examples of package installation without dependencies (Section 2.1.7), package installation with dependencies and different file types (Section 2.1.8), package removal (Section 2.1.9), and package update (Section 2.1.10).

These smaller examples fit into an overall process. The end goal of this process is for the administrator to change the packages installed on the systems she manages. The steps are:

1. The administrator uses the GUI (Section 2.1.1) or the command-line tools (Section 2.1.2) to perform package-based configuration management. This process uploads files to the repository.
2. The repository makes the uploaded files available for download (Section 2.1.3).
3. The repository notifies the systems that updated files are available (Section 2.1.4).
4. The systems receive the notification and download the updated files

(Section 2.1.5).

5. If the administrator's package instructions pertain to the system, the package instructions are enacted (Section 2.1.6).

6. Depending on the administrator's package instructions, the systems may enact package installation (Sections 2.1.7 and 2.1.8), removal (Section 2.1.9), or update (Section 2.1.10).

2.1.1 GUI Package-based Configuration Management

Consider a system administrator that manages thousands of clients at several sites around the globe. The system administrator has just finished testing a new security release for a fictional package `foobar`. She decides to have all systems that are used for testing update `foobar`. The administrator clicks on a GUI to add the package to the "testing group" (explained later), then clicks to synchronize with the repository. Within minutes all of the testing systems that are online have followed her package instructions and have updated the `foobar` package. As an offline testing system comes online, the system automatically updates its copy of `foobar` as instructed.

2.1.2 Command Line Package-based Configuration Management

A similar motivation with the same result as the previous example involves software testing. To test software, a developer builds the software and then tests it on a group of systems. To automate this process, whenever the developer builds the software the software is automatically deployed to the systems she has been allocated for testing.

2.1.3 Repository Uploads

Administrators need the ability to upload packages and other metadata such as package instructions to a repository. In this example, an administrator uploads a package and some metadata to a Stork package repository. The repository makes the data available for download by the clients.

While this example focuses on a single administrator, mutually distrustful administrators may securely share a Stork repository. This is described in Chapter 3.

2.1.4 Pushing Repository Notifications

Clients must have current metadata from the repository to perform actions such as installing a package or enacting the administrator's package instructions. All Stork clients that use a repository subscribe to a publish-subscribe channel for that repository. Whenever the repository's contents change, the repository pushes out a file that describes the change using the publish-subscribe channel.

2.1.5 Selective Content Updates

A repository stores different types of files. There are packages, package metadata, and several other types of metadata (described in detail later). Clients must have updated metadata to follow package instructions. However, in addition to a client knowing an update has occurred, it is helpful to know what type of metadata has changed. A stork client can detect which type of metadata on the repository is different from its local metadata and download only that type of metadata. This helps to prevent clients from downloading extraneous information.

2.1.6 Enacting Package-based Configuration Management Instructions

A Stork client enacts the package instructions from the administrator. Administrators may be selective with regards to which clients apply the package instructions (for example, only installing a package on a subset of clients). Each client uses the administrator's package instructions to decide which of the package instructions apply to it. The client then performs the package actions for those package instructions.

2.1.7 Installing a Package

Suppose that an administrator wants to install a fictional package `foobar` on a system. She is unsure what other software `foobar` may need to run. She requests the installation of `foobar` and the necessary packages are downloaded from the repository and installed.

2.1.8 Multiple Package Types and Dependencies

There are different package formats used by different package installers. For example, RPM installs RPM packages and `tar` is used for tarballs. The package

formats are also distinctly different with tarballs being a simple archive of files and RPMs having dependency information, an optional signature, and other information about the package.

In different situations either tarballs or RPMs are more appropriate. For instance, if a distributor is packaging a complex piece of software with multiple dependencies for a wide variety of scenarios, the RPM format seems more appropriate than a tarball. However, if an administrator wants to quickly and easily deploy a script to all of the machines she manage, using a tarball seems more appropriate.

Stork supports multiple package types allowing a packager to choose the type of package that is best suited for the software she is deploying. An administrator can use RPMs for some tasks and tarballs for others and the packages are both managed using Stork. As a result, Stork allows RPM dependencies to be met using tarballs which provides better flexibility and functionality in dependency resolution.

2.1.9 Removing a Package

In addition to installing packages, an administrator may also remove packages using Stork. This removes the software from the system cleanly.

2.1.10 Updating a Package

Installation adds a package if no version of the package exists. However, it is common for an administrator to want to update the existing version of a package to a newer version. Stork supports the update of packages to serve this purpose.

2.2 Component Overview

Stork consists of four main components:

- *Configuration management tools* for packaged-based configuration management. These allow an administrator to control many clients collectively, indicate trust in packages, and upload packages and metadata to the repository.
- A *repository* that serves packages and metadata to clients. The repository is designed so that administrators that do not trust each other can share a repository.
- A set of *client tools* that are used in each client to manage its packages by interacting either directly with the repository or through the nest.
- A *nest* that coordinates sharing between VMs. The nest avoids duplicate downloads and shares packages between VMs on the same physical machine.

The package-based configuration management tools are used by an administrator to control the packages installed on the administrator's clients by creating package instructions. The package-based configuration management tools are usually run on the administrator's desktop system (which may or may not be managed using

Stork) and allow the administrator to easily manage clients remotely. The administrator can manage a collection of clients under her control using an abstraction called a *group*. For example, an administrator can arrange some of her clients into a group and then specify that a group should install a specific package. This will cause all clients in that group to install the package. The package-based configuration management tools create files that contain the package instructions and describing what package actions a client should take. These files are made available to the clients by uploading them to a package repository.

A Stork package repository hosts the package instructions that describe what package actions a client should perform. A Stork package repository also hosts packages, package metadata, and files that are used to indicate trust in packages and entities. A Stork package repository is designed to host content from administrators of different clients that may be mutually distrustful (described in detail in Chapter 3). When information is uploaded to the repository, the repository sends out a notification to the clients it serves using a publish-subscribe system.

Each client runs the client tools component of Stork. The client tools manage packages and listen for notifications from the repository. Upon receiving a notification the client updates the local copy of the repository metadata by downloading any new metadata. If there are package instructions from the administrator that manages this client, the client tools perform the requested

package action. An administrator can also manually manage packages on a client using the client tools. Packages may be installed, removed, or updated. When a package is installed or updated, the client tools decide which packages are needed to meet the dependencies of the package requested by the administrator. These packages are then downloaded and installed.

In some cases the clients are VMs on the same physical machine. In this case, the nest runs on the physical machine and acts as a cache for downloaded files. When the client tools want to download a file, the client tools do not contact the repository directly. Instead the client tools ask the nest for the file. If the nest has not downloaded the file, the nest downloads the file and places the file in a cache. When a file that is requested is in the cache, the nest shares that file with the VM. When the client tools install a package, the client tools ask the nest to unpack the package and share the files in the package. This way, if client tools in another VM install the same package, the files in the package are securely shared between the VMs by the nest. The nest is described in detail in Chapter 4.

Figure 2.1 shows the different components in Stork. The leftmost administrator uploads a package to a package repository. The middle administrator is uploading files that have package instructions for the clients she manages. The rightmost administrator is logged directly into a VM and is managing the VM using the client tools. The repositories in the middle of the diagram notify the clients they serve of

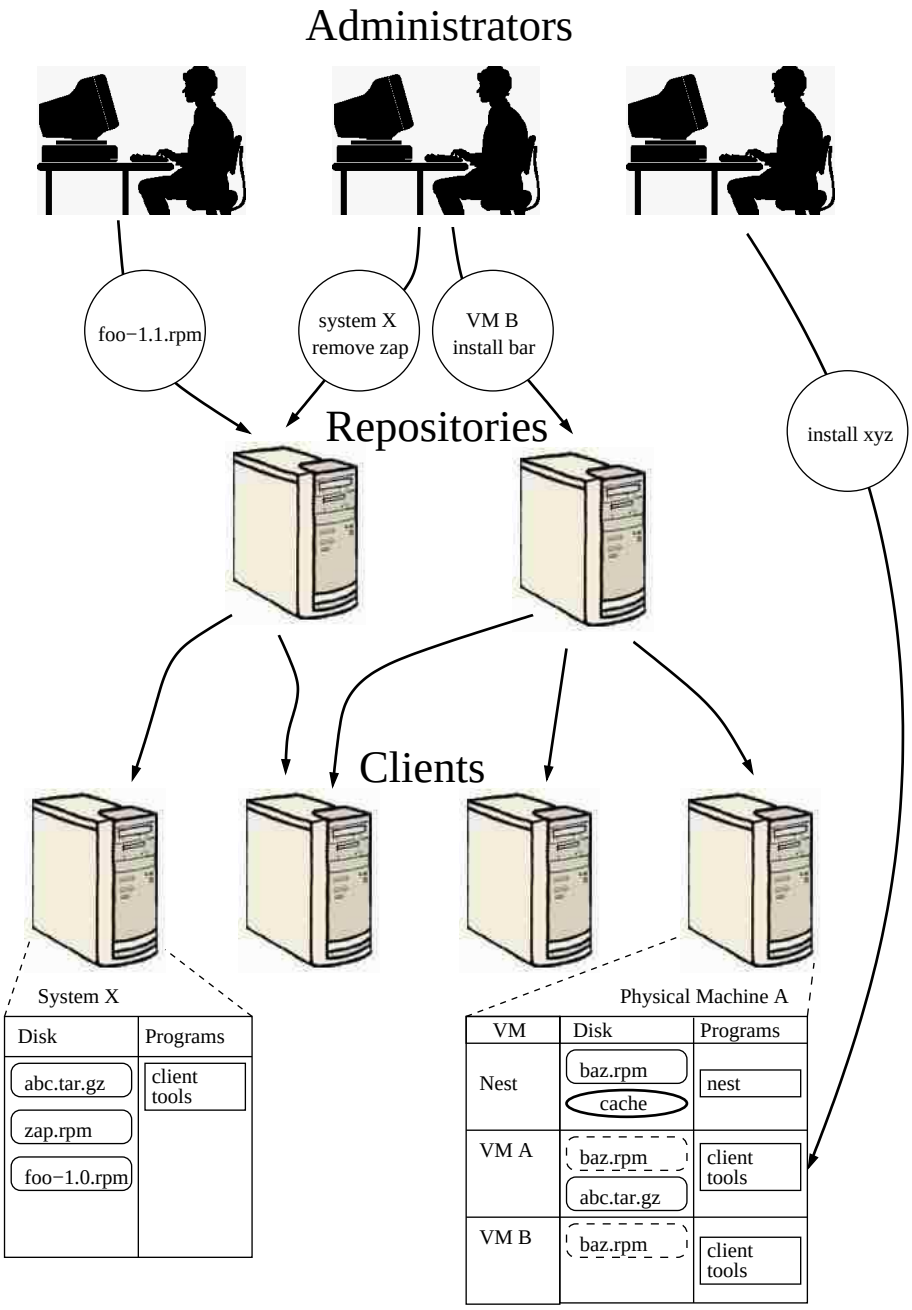

FIGURE 2.1. **Stork Overview.** Clients (bottom) install packages using the stork client tools. A VM (bottom right) installs packages with the client tools in a shared manner by using the nest on the same physical machine. Repositories (middle) host the packages and metadata created by the administrators (top).

updated content by pushing out a message via a publish-subscribe system. The clients at the bottom of the figure will download the updated information from the repository. The client on the left is a system that is running the client tools and has several packages installed. The client of the right hosts multiple VMs on its physical machine. VM A and VM B have both installed a package `baz.rpm` through the nest and so it is shared between both VMs. The nest has a cache that it uses to ensure that duplicate downloads do not occur. The nest may either be in its own VM or inside the VMM (this is explored in Chapter 4).

These components are now described in detail.

2.3 Configuration Management Tools

The package-based configuration management tools allow an administrator to manage her clients without contacting the clients directly. Client management is performed by the creation of different types of files (described in Section 2.3.1). Package-based configuration management is done with the aid of an abstraction called a group (Section 2.3.2). There are two files that contain the package instructions used to manage groups of clients: `groups.pacman` (Section 2.3.3) and `packages.pacman` (Section 2.3.4). These files are created by the tools for package-based configuration management: the Stork GUI and the command line tool `storkutil`. The GUI (Section 2.3.5) is mostly geared toward administrators that want to quickly get up and running and don't mind that some of the functionality (and complexity) of Stork is hidden from them. `storkutil` (Section 2.3.6) is meant to be used on the command line and is more appropriate for scripts and power users.

2.3.1 Stork File Types

This section provides a quick overview of the types of files Stork uses. The majority of these files can be created by the package-based configuration management tools. Each type of file is covered in more detail in the section that describes the

File Type	Purpose
Packages (RPM, tar.gz)	The software to be installed on clients
Package Metadata	Information about packages including dependencies
Public Key	Used to verify the authenticity of signed files
Private Key	Used to sign a file to testify to its authenticity
Master Configuration File	Configuration file for the Stork client tools
Trusted Packages (TP)	Specifies what packages and entities an administrator trusts
Pacman Groups	Specifies which clients belong to which groups
Pacman Packages	Specifies which packages the groups of clients should install

TABLE 2.1. **Package-based configuration management files:** This table shows the different types of files managed by the package-based configuration management tools. The type of file and purpose in Stork are described.

component that uses the file. An overview of this section is shown in Figure 2.1.

The *packages* (for example, the `foobar.rpm`) contain the software that may be installed on the client. Stork supports packages in RPM and tarball format.

Package metadata describes a package. The client uses the package metadata to get information about the available packages without downloading the packages. Package metadata may include a list of URLs that point to the package. This allows a Stork repository to support packages that are hosted on other servers.

An administrator's *public key* is distributed to all of the clients she administers and is used by the client tools. The corresponding *private key* is used by the

administrator to sign a file to testify to its authenticity. Each repository also has a public key it gives to the clients that use the repository and a private key the repository uses to sign metadata. By checking the signature of the downloaded metadata, the client can verify that the file came from the repository. More discussion about this can be found in Chapter 3.

The *master configuration file* is a configuration file used by the client tools. The master configuration file is created by the administrator and used to control the settings of the clients she manages. It indicates things such as the transfer method, repository URLs, and types of packages to use.

The administrator's trusted packages file (*TP file*) indicates which packages the administrator considers to be candidates for installation. The TP file does not cause those packages to be installed, but instead indicates trust that the packages have valid contents and could be installed. For example, while the administrator was testing the latest release of `foobar` she could add it to her trusted packages file. Then the administrator could request the installation of the new version of `foobar` on a testing client. However, this does not mean that other clients that the administrator manages will install the new version of `foobar`. The administrator must request that the new version of `foobar` be installed before `foobar` is installed.

There are two files that contain the package instructions that are used for

package-based configuration management: `groups.pacman` and `packages.pacman`. These files are described in detail after a discussion about the underlying abstraction, a group.

2.3.2 Groups

An important abstraction for managing clients is a group. A group is a logical association of multiple clients defined by the administrator of those clients. The fact that groups are purely a logical organization is very important: the clients in a group do not need to be connected in any way and a single client can be a member of multiple groups. Groups simplify configuration management by allowing package operations to be specified on collections of clients.

There are three types of groups: simple, composite, and query result. Simple groups allow administrators to manage a collection of clients as if the collection of clients was a single client. One can declare that systems 1, 2, and 3 are part of group G and then can say that package X should be installed on group G. This results in systems 1, 2, and 3 installing package X. Simple groups can also specify that all clients of a certain type or on a certain physical machine belong to a group. There is no limit to the number of clients in a group.

Composite groups are created by performing an operation on existing groups. The

supported operations for composite groups are UNION, INTERSECT, COMPLEMENT, and DIFFERENCE. Composite groups do not have clients directly included. Instead, the clients in the group depend on the operation and the membership of the other groups. For example, composite group H may be defined as the UNION of group I and group J and will contain all of the clients appearing in either group I or group J. Groups I and J can be any types of groups, including composite groups.

Query result groups are computed based on some property of the clients. PlanetLab is a major user base of Stork and therefore the management tools provide support for building groups from the result of CoMon [59] queries. For example, a group contains all physical machines with more than 1GB free disk space could be created using the CoMon query `select='gbfree > 1'`.

Query result groups are evaluated when the group is created. The list of machines returned by the CoMon query is created as a simple group. This means that only simple and composite groups are written to the underlying files.

2.3.3 `groups.pacman`

The `groups.pacman` file is the first part of the package instructions and defines which clients belong to which groups. Simple groups can select clients based upon

different attributes. The `NAME` attribute selects clients by hostname. The `SLICE` attribute selects VMs by VM name. Attributes accept wild cards so that `SLICE="ABC*"` will select any VM whose name begins with ABC.

Composite groups are written with a tag that indicates the operation to be performed and the other groups to use as arguments. Composite groups may not select clients by attribute because individual clients cannot be added to a composite group directly. The four types of composite groups are `UNION`, `INTERSECT`, `DIFFERENCE`, and `COMPLEMENT`.

Figure 2.2 shows two simple groups `Test` and `Production`. `Test` contains all of the clients that are on two physical machines. `Production` contains all VMs named `VM_test`. There is also a union composite group `Both` that contains the all clients in either `Test` or `Production`.

2.3.4 `packages.pacman`

The `packages.pacman` file is the second part of the package instructions and specifies what package actions the clients in a group should perform. The supported package actions are to update, install, or remove packages. This allows administrators to specify package instructions that will be carried out on groups of clients. For example, the administrator may specify that a particular package

```
<GROUPS>
<GROUP NAME="Test">
<INCLUDE NAME="planetlab1.arizona.net"/>
<INCLUDE NAME="planetlab2.arizona.net"/>
</GROUP>

<GROUP NAME="Production">
<INCLUDE SLICE="VM_test"/>
</GROUP>

<UNION NAME="Both" GROUP1="Test" GROUP2="Production"/>
</GROUPS>
```

FIGURE 2.2. **Example groups.pacman** There are two simple groups `Test` and `Production` and one composite group `Both`. `Test` contains all of the clients that are on two physical machines. `Production` contains a VM named `VM_test`. The composite group `Both` contains the all clients either in `Test` or `Production`.

```
<PACKAGES>
    <CONFIG GROUP="Both">
        <INSTALL PACKAGE="emacs" VERSION="2.2"/>
        <REMOVE PACKAGE="vi"/>
    </CONFIG>
    <CONFIG GROUP="Test">
        <UPDATE PACKAGE="foobar"/>
    </CONFIG>
    <CONFIG>
        <UPDATE PACKAGE = "firefox"/>
    </CONFIG>
</PACKAGES>
```

FIGURE 2.3. **Example packages.pacman** VMs in the group `Both` will install `emacs` version 2.2 and remove `vi`. VMs in the group `Test` will update `foobar` to the latest version. All clients under the administrator's control will update `firefox`.

should be installed on all VMs on a physical machine, while another package should only be installed on clients used for a specific task (and thus in a specified group).

Figure 2.3 shows an example `packages.pacman` file. This file specifies package instructions for the groups Both and Test. The final action does not specify a group and is performed by any client using this file.

Using the files in Figures 2.2 and 2.3 as an example, if the client is a VM called `VM_test` and so is in the `Production` and `Both` groups, the client will install `emacs` version 2.2, remove `vi`, and update `firefox`.

2.3.5 GUI

The GUI provides common package-based configuration management functionality. This includes operations such as selecting groups of clients, requesting the installation of packages on clients, and trusting new packages. The GUI also has some features to better support administrators that manage VMs on PlanetLab. The GUI doesn't contain the actual code to do the cryptographic operations or create files but instead calls functionality in `storkutil`.

When started, the GUI asks for a username and password that will be used to authenticate with the repository, as well as the location of a private key (with optional password). The private key is used to sign configuration files. The corresponding public key is what the clients use to verify that the signatures on downloaded configuration files are legitimate.

After the GUI verifies the repository login information and the validity of the private key, the administrator can begin configuring her groups and the package actions for those groups. The GUI retrieves the latest package instructions from the repository and displays any groups and package actions that were previously defined. The administrator can add a new group, by clicking on the "add group" button and giving the group a name. Then the administrator may proceed to either add clients to form a simple group or form a composite group by choosing an

operation and source groups.

In addition to creating, modifying, and deleting groups, an administrator may define package actions for a group; that is, specify packages that should be installed, updated, or removed for all clients in the group. To install or update a package, the administrator may specify the name of the package and let the clients find a version of the package that is trusted. Alternatively, the administrator may provide a local copy of the package and then the GUI will add trust of this package to the administrator's TP file and upload the package to the repository.

If any changes are made in the GUI, an icon shows that the changed package instructions must be uploaded to the repository for the changes to have effect. The administrator can then sync with the repository, which means that the new package instructions will be uploaded to the repository. If a new package has been added, the administrator's TP file and the package will also be uploaded.

In order to provide better support to administrators of VMs on PlanetLab [62], the GUI interacts with PlanetLab's central database. A PlanetLab administrator accesses VMs in collections called *slices*. The GUI retrieves information about the slices the administrator has access to and the public keys for each of these slices from PlanetLab's central database. When the administrator requests that a VM is added to a slice, the GUI contacts PlanetLab's central database to request the VM

addition. The GUI also sets the `startscript` (described in Section 2.5.5) to bootstrap the client tools on the VMs. This allows an administrator to interact solely with the Stork GUI once her PlanetLab account is enabled, rather than needing to go back and forth between PlanetLab's website and the Stork GUI.

2.3.6 `storkutil`

The command line tool `storkutil` also provides the functionality needed for package-based configuration management of clients. `storkutil` is commonly integrated into the build process of software to automate the creation of package instructions.

Using `storkutil` also allows finer control over how actions are performed compared with the GUI. For example, an administrator that uses the GUI and wants to add packages has the individual packages added to her TP file. Administrators that use `storkutil` may instead choose to delegate trust to other entities for sets of packages.

`storkutil` generates, signs, and updates the necessary files based upon the administrator's request. For example, to add a new group named `EXAMPLE_CLIENTS` with every VM on `A.example.com`:

```
storkutil pacgroups include EXAMPLE_CLIENTS A.example.com
```

Then, to add more VMs to the same group:

```
storkutil pacgroups include EXAMPLE_CLIENTS B.example.com
C.example.com D.example.com E.example.com
```

Finally, to say that all of the clients in the new group should install the package `foobar`:

```
storkutil pacpackages group EXAMPLE_CLIENTS install foobar
```

The command line tool `storkutil` takes care of updating the administrator's files as well as signing the files with the administrator's private key. If the file does not exist, `storkutil` creates it. The one thing that the GUI does that `storkutil` doesn't is upload the files to the repository. Uploading the files to the repository is something that can be scripted with other tools, if desired (for example, using `curl`).

2.4 Repository

The basic purpose of a Stork repository is to receive uploaded files from administrators and serve those files to clients. It is important to understand how files are protected from tampering (Section 2.4.1), to understand how the repository handles uploaded files (Section 2.4.2). A Stork repository also provides push-based notifications (Section 2.4.3) and creates torrent files to support BitTorrent transfers (Section 2.4.4).

2.4.1 Protecting Files

This section describes Stork's security mechanisms for protecting files in enough detail to understand how the repository handles uploaded files (protection is revisited in Chapter 3). First, there is a description of how tampering is detected. Then follows a discussion of a mechanism that allows any party (including the repository) to detect tampering. This mechanism also disambiguates files with the same name but different contents.

A Stork repository hosts different types of files that are protected and stored in different ways (as Figure 2.2 shows). Stork uses several mechanisms to detect if an attacker has tampered with files. The two basic mechanisms are public/private key

File Type	Purpose	Protection	Tarball
Master Configuration File	Configuration file for the Stork client tools	Signed and Embedded Public Key	Yes
Trusted Packages (TP)	Specifies what packages and entities an administrator trusts	Signed and Embedded Public Key	Yes
Pacman Groups	Specifies which clients belong to which groups	Signed and Embedded Public Key	Yes
Pacman Packages	Specifies which packages the groups of clients should install	Signed and Embedded Public Key	Yes
Packages (RPM, tar.gz)	The packages to be installed on clients	Embedded Secure Hash and Immutable	No
Package Metadata	Information about packages including dependencies	Embedded Secure Hash and Immutable	Yes
Root Repository Metadata	Information about what content is on the repository	Signed (Repository)	No

TABLE 2.2. **Files hosted by Stork repositories:** This table shows the different types of files hosted by Stork repositories. The type of file, purpose in Stork, and how the file is protected are described. In some cases files are protected by the signature of a private key. In other cases, the requester knows the secure hash of the file when requesting it and can check that its contents match. In addition, the file may be treated as immutable by the repository and so a non-malicious repository will not overwrite it. Many of the files are placed together with the other files of the same type in a single tarball.

pairs and secure hashes. Perhaps the easiest way to understand why different protection mechanisms are needed is to look at two examples of types of files a repository serves: a package and a `groups.pacman` file. A package is a file that does not change. An administrator may upload newer versions (which are different files) but the contents of an existing package do not change as a result. Since a package is immutable, a secure hash is appropriate for protection. To see why, suppose there is a client that knows the secure hash of the contents of a given package. If the client tries to download that package one month later the client can know that the package has not been tampered with by checking the secure hash. The secure hash of a file that does not change serves as a way to verify the contents for all time.

In contrast, consider an administrator's `groups.pacman` file. This file is updated by the repository whenever an administrator changes the groups that clients belong to. A client that knows the secure hash of the contents of an earlier version of the `groups.pacman` file cannot use this information to validate a new version. This is because the administrator's `groups.pacman` changes, so the secure hash changes. Instead of using a secure hash for protection, the administrator's `groups.pacman` is signed by the administrator's private key. The clients can use the administrator's public key to check if the file has been tampered with. The rationale behind this will be discussed in Chapter 3.

Signatures and secure hashes are useful to detect tampering. However they require

the party that wishes to check for tampering to know the secure hash or corresponding public key. Stork embeds the secure hash or public key in the file name or URL (as in self-certifying path names [48]). Embedding the secure hash or public key in the file name or URL allows any party to check if a file has been tampered with. This also allows a party to request files that can be verified using a specific public key or files that have a specific secure hash. The ability to request a file with a specific secure hash disambiguates files with the same name but different contents.

Packages and package metadata are stored on a Stork repository at URLs containing their secure hashes. These items are considered to be immutable. Any party can verify that a file has not been tampered with by checking the secure hash of the retrieved data. This allows a Stork repository to check that uploaded packages and package metadata is valid. This is also used by the client to validate that information retrieved from a Stork repository is correct.

Several files in Stork have public keys embedded in their names instead (including the master configuration file, TP file, `groups.pacman` file, and `packages.pacman` file). Using an embedded public key, a Stork repository can verify the file is unmodified by checking that the file is correctly signed and that the public key in the file name corresponds to the private key that signed the file.

Some files are not immutable and may be changed by the administrator. Administrators may upload an updated file to a Stork repository and the file will be included if it is correctly signed. Replaying outdated files is prevented by a timestamp in the signed file.

More details about the protection of files in Stork is described in Chapter 3.

2.4.2 Handling Uploaded Data

A package repository's main task is to serve files much like a normal web server. A package repository differs from a web server in that it serves up packages and metadata instead of web content. However, a Stork repository is also different from other package repositories in the way it handles uploaded content. A Stork repository is intended to host content uploaded by many different administrators. These administrators need not cooperate or even trust each other not to be malicious. Packages and metadata are accessed in a way that allows the repository to be shared without implicitly trusting other administrators (described in detail in Chapter 3). This leads to design differences with other package repositories including how different types of uploaded data are handled.

It is important to note that a client does not rely on a package repository to perform actions for its security. Even when the repository is malicious, a client has

strong security guarantees. However, the security and efficiency of a Stork client is improved by a repository that is well-behaved. As with other security details, this is deferred to Chapter 3.

Administrators may upload package metadata. The package metadata is placed in a file named by the secure hash of contents of the package metadata. This is so that any party that knows the file name of the package metadata can test the file for tampering. This file is then archived in the tarball that contains the other package metadata. This prevents clients from needing to download a separate small file for the metadata of each package.

Administrators may also upload packages to a Stork repository. The package is placed in a directory named by the secure hash of its contents, so that any party that has the URL of the package can verify that the package has not been tampered with. The package metadata is extracted from the package and is processed in the same way as uploaded package metadata.

A Stork repository also allows administrators to upload other types of files. TP, `groups.pacman`, `packages.pacman`, and master configuration files must be signed before being uploaded. A Stork repository will only store a signed file for which the secure hash in the filename matches the public key, the file's signature is verifiable by the public key, and the timestamp is newer than any existing signed file of the

same name. These checks improve efficiency and security and are described in more detail in Chapter 3. After this file has been validated it is placed into a tarball with the other files of the same type. This is also described in more detail in Chapter 3.

A Stork repository uses a file called the *root repository metadata* to indicate the current repository state. The root repository metadata is signed by the repository's private key so that clients can authenticate that the root repository metadata came from the repository. The root repository metadata consists of the secure hashes and sizes of the tarballs that contain the other types of metadata. By downloading the root repository metadata, a client can tell what content on the repository is new and download only that content. This is described in more detail in Section 2.5.2.3.

The layout of the files on a Stork repository is shown in Figure 2.4. The root repository metadata contains the locations, secure hashes, and sizes of the other tarballs. The tarballs contain the TP files, package metadata, pacman files, and master configuration files. TP files contain the secure hashes of the package metadata. The package metadata contains the secure hash of the package itself. More detail is provided about the protection of files in Chapter 3.

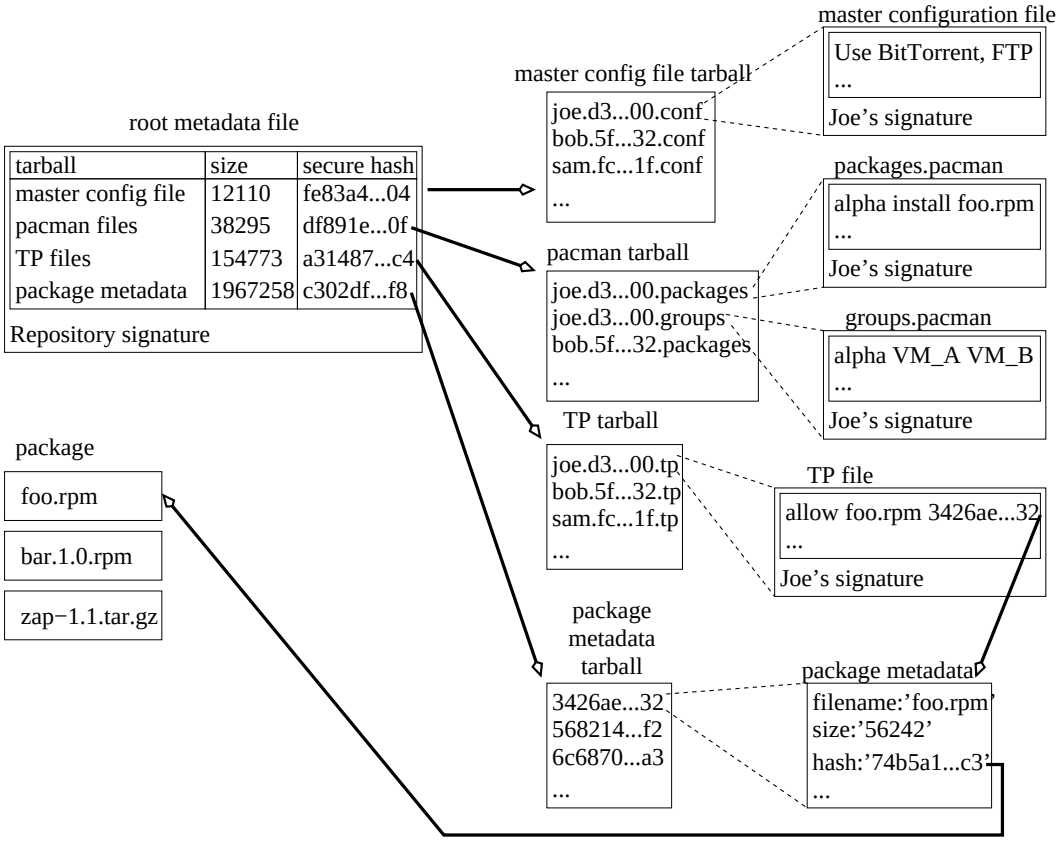

FIGURE 2.4. **Files on a Stork Repository.** This diagram represents the files on a Stork repository. Arrows show the file that the secure hash corresponds to. The contents of the files do not correspond literally to the contents of the actual files. They are edited for clarity and space.

2.4.3 Pushing Notifications

A Stork repository notifies clients when the repository contents have changed. This is done by pushing notifications through a publish/subscribe system [30]. In a publish/subscribe system, a subscriber registers its interest in an event and is subsequently notified of events generated by publishers. One such publish/subscribe system is PsEPR [12]. The messaging infrastructure for PsEPR is built on a collection of off-the-shelf instant messaging servers running on PlanetLab. PsEPR publishes events (XML fragments) on channels that clients subscribe to. Behind the scenes PsEPR uses overlay routing to route events among subscribers.

However, simply receiving a notification that content has changed does not address the important question of *what* content has been changed. This is especially difficult to address when clients may miss messages or suffer other failures. To provide clients with information about what has changed, the repository pushes out the root repository metadata. As is described in Section 2.5.2.3, this allows the client tools to download only the updated tarballs.

2.4.4 Repository Support for Efficient Transfers

A Stork repository makes its packages and tarballs containing the metadata available for download through HTTP. However, Stork also supports other transfer types to allow an administrator to use the transfer protocol she prefers. Some transfer mechanisms are simple for the repository to handle (like CoBlitz and Coral which require no special handling) and others (like BitTorrent [22]) require modifications.

To support BitTorrent downloads, a Stork repository runs a BitTorrent tracker and a modified version of the `btlaunchmany` daemon provided by BitTorrent. The `btlaunchmany` daemon monitors a directory for a new or updated package or tarball. When a new file is uploaded to a Stork repository it is placed in the monitored directory. The `btlaunchmany` daemon creates a torrent file and seeds the new file. Unique naming is achieved by appending the secure hash of the seeded file to the name of the torrent. The secure hash must be added because the tarballs do not have the secure hash in the file name and the secure hash of package is in the directory path, not the package itself. The torrent file is placed in a public location on a Stork repository for subsequent download by the clients through HTTP. Once a client has downloaded the torrent, the client may use BitTorrent to download the file.

2.5 Client Tools

The purpose of the client tools is to manage packages, enact package actions based upon package instructions, and listen for notifications from the repository. The client tools can be run via administrator package instructions or be run manually by an administrator.

The client tools are comprised of `stork`, `pacman`, and `stork_receive_update`. The `stork` tool performs dependency resolution, downloads packages from the repository, and installs, updates, and removes packages. `pacman` (not to be confused with the package manager Pacman [7] for Arch Linux) is responsible for enacting the package actions in the package instructions. `stork_receive_update` is responsible for receiving notifications from a Stork repository pushed out over a publish / subscribe system.

Before delving into the client tools, this section first describes an abstraction called a functionality manager (Section 2.5.1). This is important to understand because multiple components of the client tools (and the nest) use functionality managers. Then follows a description of the `stork` tool (Section 2.5.2) and the functionality managers `stork` uses to transfer files (Sections 2.5.2.3 and 2.5.2.4) and manage packages (Sections 2.5.2.5 and 2.5.2.6). The description of the client tools continues with `pacman` (Section 2.5.3) and `stork_receive_update` (Section 2.5.4). This

section concludes with a discussion of PlanetLab specific functionality that is used to bootstrap the client tools (Section 2.5.5).

2.5.1 Functionality Managers

Stork uses an abstraction called a *functionality manager* to provide extensibility. The purpose of a functionality manager is to hide the complexity of *how* something is done and instead have the programmer specify *what* the result should be. Functionality managers hide the complexity inherent in the multiple ways of performing an action (for example, multiple transfer methods for file retrieval). This means the functionality manager also must decide *which* way to perform an action. The different ways an action can be performed are implemented by *specialized plug-ins*.

For example, the transfer functionality manager may be called to download a file. The configuration options specify the preference of which specialized plug-in should be used by the functionality manager to perform the download. The specialized plug-in that was chosen may use the transfer protocol BitTorrent to download the file, for example.

Functionality managers leverage specialized plug-ins for low-level functionality. However, a functionality manager can provide more complex functionality to the

rest of the system than is provided by the specialized plug-ins it accesses. For example, the "package" functionality manager performs dependency resolution by requesting individual package information from the package specialized plug-ins. The package functionality manager decides which packages should be used to satisfy a dependency, resolves unmet dependencies of these packages, and detects conflicts. Since the package functionality manager performs dependency resolution, files in tarballs can be used to fulfill RPM file dependencies even though tarballs and RPMs are handled by separate specialized plug-ins. Handling dependency resolution within the functionality manager instead of individually in each specialized plug-in provides better interoperability among specialized plug-ins.

Functionality managers can handle failures. They detect and recover from specialized plug-in failures whenever possible. When a failure occurs in a specialized plug-in, the functionality manager utilizes other appropriate specialized plug-ins to retry the operation.

The failure handling ability of a functionality manager is useful for supporting third-party specialized plug-ins. Many of the specialized plug-ins Stork uses access prototypes created by other researchers. The research prototype code is more likely to have faults than popular commercial code [16]. A functionality manager's failure handling prevents prototype failures in a specialized plug-in from negatively impacting the administrators using Stork.

A specialized plug-in performs an action in a specific way. For example, there are transfer specialized plug-ins for BitTorrent, CoBlitz, and even for nest communication. The upper level interface for each specialized plug-in is uniform and implements only a few simple methods. For example, the transfer interface implements four methods: `init_transfer_program()`, `close_transfer_program()`, `retrieve_files()`, and `transfer_name()`. The init and close methods are called before the specialized plug-in is used the first time and after the specialized plug-in is used the last time, respectively. The `transfer_name()` method returns a string containing the name of the transfer protocol. The `retrieve_files()` method does the retrieval in a way specific to the underlying protocol the specialized plug-in implements.

The functionality managers that Stork uses are:

Transfer The transfer functionality manager is used by `stork`, `pacman`, and the nest to download files. The transfer specialized plug-ins implement a protocol for retrieving data. Transfer specialized plug-ins are responsible for retrieving a particular object given a URL for that object. There are specialized plug-ins that support transfer types including CoBlitz [57], BitTorrent [22], Coral [33], HTTP, HTTPS, FTP, and transfers through the nest. This functionality manager is described in detail in Section 2.5.2.3.

Package The package functionality manager manages packages on a client. This is used by `stork` to install, remove, and update packages as well as query information about installed packages and perform dependency resolution. Different specialized plug-ins understand different package types or perform actions in different ways. There are package specialized plug-ins that install RPM and tar package formats in an unshared manner and RPMs in a shared manner. These are described in Section 2.5.2.5 and revisited in Chapters 3 and 4.

Share The share functionality manager shares files between VMs on the same physical machine. It is used by the nest to share between clients. Share specialized plug-ins are implemented for different architectures that share files in different ways. These specialized plug-ins protect files from modification, map content between VMs, and authenticate client VMs. Currently Stork supports share specialized plug-ins for PlanetLab and Linux VServers. This functionality manager is described in detail in Section 4.7.1.

Prepare The prepare functionality manager is used by the nest to unpack packages without installing them. A package is unpacked without installation when it is being "prepared" for sharing with VMs. A prepare specialized plug-in is specific to a package type, similar to a package specialized plug-in. A prepare specialized plug-in differs from a package specialized plug-in in that package install scripts are not run and databases that track installed packages are not updated. This is

because the package is not installed in the nest, it is unpacked only to share the files within. There only exists one specialized plug-in for this functionality manager and it unpacks RPMs. Sharing tarball packages could be supported but is not at this time. RPM sharing was implemented before tarball sharing because RPMs are used in 89% of the packages installed by multiple administrators. The prepare functionality manager is described in more detail in Section 4.7.3.

2.5.2 `stork`

This section describes the `stork` package manager. `stork` can be run from the command-line by an administrator. The syntax of `stork` is similar to `apt` [5] and `yum` [92]. `stork` is a package manager that enacts package actions securely.

Section 2.5.2.1 describes the master configuration file used by `stork`. Section 2.5.2.2 discusses the overall process of how package installation, update, and removal work at a high level. The steps used are dependency resolution and trust verification (described in detail in Chapter 3), data transfer by the transfer functionality manager (Sections 2.5.2.3 and 2.5.2.4), and package actions by the package functionality manager (Sections 2.5.2.5 and 2.5.2.6).

```
# Lots of logging
veryverbose
noplckeys

bittorrenttrackerhost=stork-repository.cs.arizona.edu

bittorrenttrackerport=6880
bittorrentuploadrate=0
bittorrentseedlookuptimeout=30

repositorypackagedir = *_packages_*
repositorypackageinfo = stork-repository.cs.arizona.edu/packageinfo

tagprefrules=%ARCH%<PlanetLabV4<PlanetLabV3

username = exampleadministrator
publickeyfile = /usr/local/stork/var/keys/default.publickey
packagemanagers = nestrpm, rpm, tar
transfermethod= nest,bittorrent,coblitz,coral,http,ftp
nestport=648
```

FIGURE 2.5. **Example master configuration file.** This file is an example master configuration file for the stork client tools. All of the client tools (`pacman`, `stork`, and `stork_receive_update`) use the same configuration file. The options specified here do things like set up BitTorrent, change the verbosity level, choose the repository, and choose the transfer methods and package types that are supported. The signature is removed for clarity.

2.5.2.1 Master Configuration File The behavior of `stork` is in large part controlled by the master configuration file. The master configuration file contains the configuration settings for `stork`, `pacman`, and `stork_receive_update`. One of the main reasons a single configuration file is used by different tools is so that the common functionality used by the client tools has the same settings. For example, the transfer functionality manager's preferences for transfer protocol should be used by all of the client tools that transfer files.

Figure 2.5 shows an example master configuration file. The options determine the repository to use, the verbosity of output, and the transfer protocol to use (among other things). The functionality managers use the options specified in the master configuration file to determine which specialized plug-ins to try first. The example master configuration file shows the `transfermethod` option set to nest, bittorrent, coblitz, coral, http, ftp. This is the order in which the transfer functionality manager uses these specialized plug-ins.

2.5.2.2 Package Actions `stork` installs, updates, and removes packages (collectively these are called *package actions*). To perform package actions, `stork` begins by reading the master configuration file and updating metadata from the package repositories. If there is a new master configuration file after the client updates the metadata from the package repository, `stork` repeats this process.

`stork` obtains a consistent set of metadata by building a customized repository view (described in Chapter 3). Once `stork` has current and consistent package metadata, TP files, and a master configuration file, `stork` has the necessary information to perform package actions.

Install: `stork` performs several steps to install a package. Installation of a package involves *dependency resolution*, *trust verification*, *download*, and *installation*.

The first step is dependency resolution. This involves finding a set of packages to install so that that every package has all of its dependencies fulfilled. Dependency resolution is performed by the package functionality manager, which is described in Section 2.5.2.5.

The process of resolving dependencies is intertwined with trust verification. Candidate packages that are not trusted will be dismissed from consideration. Trust is specified through the use of a trusted packages (TP) file. Figure 2.6 shows an example TP file. This file specifically allows `emacs-2.2-5.i386.rpm`, several versions of `foobar`, and `customapp-1.0.tar.gz` to be installed. Each package listed in the TP file includes the hash of the package metadata for the package. Only a package whose metadata matches the secure hash of a line in the TP file can be installed. The TP file trusts the `stork` entity to know the validity of any packages that start with "stork" or "arizona". The TP file also trusts the `planetlab-v4`

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes" ?>
<TRUSTEDPACKAGES>

<!-- Trust some packages that the administrator specifically allows -->
<FILE PATTERN="emacs-2.2-5.i386.rpm"   HASH="aed...732" ACTION="ALLOW"/>
<FILE PATTERN="foobar.rpm" HASH="16b...470" ACTION="ALLOW"/>
<FILE PATTERN="foobar.rpm"  HASH="394...7fd" ACTION="ALLOW"/>
<FILE PATTERN="customapp-1.0.tar.gz"    HASH="234...341" ACTION="ALLOW"/>

<!-- Allowing the 'stork' entity lets stork packages be installed -->
<USER PATTERN="stork*,arizona*" USERNAME="stork" PUBLICKEY="RFw...eS" ACTION="ALLOW"/>
<!-- Allow access to the planetlab Fedora Core 4 packages -->
<USER PATTERN="*" USERNAME="planetlab-v4" PUBLICKEY="MAw...AQ" ACTION="ALLOW"/>

</TRUSTEDPACKAGES>
```

FIGURE 2.6. **Example TP File.** This file specifies what packages and entities are trusted. Only packages allowed by a TP file may be installed. FILE actions are used to trust individual packages. USER actions allow hierarchical trust by specifying a entity whose TP file is included. Keys and hashes are shortened for readability. The signature is contained in an XML layer that encapsulates this file and is not shown.

entity to know the validity of any package (this entity has a list of hashes of all of the Fedora Core 4 packages). More information about how TP files provide security in package management can be found in Chapter 3.

Once satisfactory trusted candidates have been found to resolve all of the dependencies, `stork` downloads the packages from a repository. Downloading is done by the transfer functionality manager described in Section 2.5.2.3.

After the packages are downloaded, the packages are installed using the package functionality manager described in Section 2.5.2.5. The packages must be installed in a specific order if there are dependencies. If package A has a dependency on package B, then package B must be installed before package A. This is important for two reasons. First, if package installation fails it leaves the packages on the client in a state in which there are no installed packages with unresolved dependencies. Second, package A may have an install script that runs a program that may need package B to be installed to work correctly.

Remove: Package removal is much less complex than installation. Before removing a package, the package functionality manager first checks to see if other packages depend upon the package to be removed. If there are dependencies that would be broken by removal of the package, then `stork` reports the conflict and exits. `stork` removes an installed package by deleting the package's files and running the

uninstall scripts for the package.

Update: Updating replaces an existing package with the preferred version (usually the most recent). If the package is not installed, update operates identically to installation. If the package is already the preferred version, then update reports this and exits. A package is updated to a preferred version by a process identical to installation until the step when the package installation is set to occur. At that time, the existing version is removed and the preferred version is installed (with the appropriate uninstall and install scripts executing). This updates the existing package with the preferred version.

2.5.2.3 Transfer Functionality Manager The transfer functionality manager downloads files from a Stork repository. The transfer functionality manager hides the complexity of different transfer protocols. It automatically fails over to other transfer protocols when failures occur. The transfer functionality manager also provides a function to synchronize the files in a directory on the repository with a directory on the client. This means that any file on the repository that differs from the client's version will be downloaded. This functionality is provided in the transfer functionality manager using the file transfer interface provided by a transfer specialized plug-in.

Each transfer specialized plug-in implements a `retrieve_files` function that takes

several parameters including the name of the repository, source directory on the repository, a list of files, and a target directory to place the files in. A successful call to `retrieve_files` returns a list of the files that were successfully retrieved. An exception is thrown when a failure occurs. The exception indicates which files the specialized plug-in has correctly retrieved and which have failed.

The order of preference of transfer specialized plug-ins is specified via an ordered list in the master configuration file. The transfer functionality manager always starts by trying the first transfer specialized plug-in in the list. If this transfer specialized plug-in should fail or return an invalid file, then the transfer functionality manager moves on to the next transfer specialized plug-in in the list. The result is that the transfer functionality manager downloads files using the preferred transfer specialized plug-in in the common case, and fails over to other specialized plug-ins when necessary.

In addition to retrieving files, the transfer functionality manager builds upon the transfer specialized plug-ins to support the synchronization of entire directories. Directory synchronization mirrors a directory hierarchy from the repository to the client. The directory synchronization functionality is used by the client tools to obtain all of the metadata tarballs from the repository.

To synchronize the metadata tarballs, the `arizonatransfer` functionality manager

obtains the current root repository metadata (the file that the repository publishes periodically using PsEPR). The root repository metadata contains a list of the metadata tarballs that comprise the repository's current state and the secure hashes and sizes of those tarballs. The `arizonatransfer` functionality manager compares the secure hashes in the root repository metadata file to those of the most recent copies of these tarballs on disk. If a secure hash does not match, then the tarball is downloaded using a transfer specialized plug-in.

2.5.2.4 Transfer Specialized Plug-ins There are seven transfer specialized plug-ins: CoBlitz [58], BitTorrent [22], Coral [33] HTTP, FTP, HTTPS, and nest. The first six transfer specialized plug-ins communicate with the repository to retrieve data and are described below. The nest specialized plug-in is used by VMs and communicates with the nest instead. If the nest has already transferred a file, the file is shared with the VM instead of being downloaded again. The nest specialized plug-in is described in Chapter 4.

CoBlitz The CoBlitz specialized plug-in retrieves files using CoBlitz. CoBlitz uses a content distribution network (CDN) called CoDeeN [85] to support large files transfers without modifying the client or server. Each system in the CDN runs a service that is responsible for splitting large files into chunks and reassembling them. This approach not only reduces infrastructure and the need for resource

provisioning between services, but can also improve reliability by leveraging the stability of the existing CDN. CoBlitz demonstrates that this approach can be implemented at low cost, and provides efficient transfers even under heavy load.

Coral The Coral specialized plug-in uses the Coral peer-to-peer content distribution network that consists of volunteer sites that run CoralCDN. The CoralCDN sites automatically replicate content as a side effect of accessing it. A file is retrieved via CoralCDN simply by making a small change to the hostname in an object's URL. Then a peer-to-peer DNS layer transparently redirects browsers to nearby participating cache systems, which in turn cooperate to minimize load on the origin web server. One of the system's key goals is to avoid creating hot spots. It achieves this through Coral [33], a latency-optimized hierarchical indexing infrastructure based on a novel abstraction called a *distributed sloppy hash table* (DSHT).

BitTorrent The BitTorrent specialized plug-in uses BitTorrent to retrieve files. BitTorrent identifies content by URL and is designed to integrate seamlessly with the web. Its advantage over HTTP is that systems that download the same file simultaneously also upload portions of the file to each other. This greatly reduces the load on the server and increases scalability. Nodes that upload portions of a file are called *seeds*. BitTorrent employs a *tracker* process to track which portions each seed has and helps a client locate seeds with the portions it needs. BitTorrent balances seed loads by having its clients preferentially retrieve unpopular portions,

thus creating new seeds for those portions.

HTTP, **HTTPS**, and **FTP** The HTTP, HTTPS, and FTP specialized plug-ins communicate using the popular web protocols.

nest A VM may download files using a shared component called the nest. This is described in Chapter 4.

2.5.2.5 Package Functionality Manager The package functionality manager performs operations on packages while hiding the differences between package types. The package functionality manager also implements dependency resolution.

Dependency resolution is necessary because a requested package must have its dependencies resolved before it can be installed. Dependencies exist because a package relies on functionality in other packages. For example, consider the installation of the `foobar` package. Suppose `foobar` depends on the installation of a few other packages, such as `openssl` and `glibc`, to operate correctly. In order to install `foobar`, the `stork` tool must determine whether `openssl` and `glibc` are already installed on the client and if not, locate candidate versions that satisfy the dependencies. The candidate versions of those packages may themselves have dependencies that must be installed. Those dependencies are resolved in the same way as the dependencies for the requested package until either all dependencies are

met or it is determined that there is a dependency that cannot be met. Dependency resolution has security implications and is described in more detail in Chapter 3.

2.5.2.6 Package Specialized Plug-ins This section describes the different package specialized plug-ins used by Stork. There are two plug-ins that install packages in an unshared manner that are described below. The plug-in that installs packages in a shared manner is described in Chapter 4.

`stork_rpm`. Stork uses the `rpm` tool to install RPM packages. The RPM database is maintained internally by the `rpm` package installer, and Stork's RPM package specialized plug-in uses `rpm` to query the package database and to execute the `install`, `update`, and `remove` package actions. All of the scripts that run when packages are added or removed are run by `rpm` itself.

`stork_tar`. Tar packages are treated differently because Linux does not maintain a database of installed tar packages, nor is there a provision in tar packages for executing install and uninstall scripts. The `stork_tar` specialized plug-in allows administrators to bundle four scripts, `.preinstall`, `.postinstall`, `.preremove`, `.postremove` that are executed at the appropriate times during package installation and removal. To support querying existing packages and removing packages, `stork_tar` maintains a package database that contains the names and versions of the tar packages that are installed. This package database mimics the RPM

database provided by the `rpm` tool.

2.5.3 `pacman`

The second component of the client tools is `pacman`. `pacman` runs in the client and supports package-based configuration management. `pacman` enacts the changes on the client that the administrator requested using the package instructions. `pacman` does this through the following steps:

1. `pacman` uses the transfer functionality manager to update the metadata tarballs from the repository.

2. `pacman` uses the `groups.pacman` and `packages.pacman` files that contain the administrator's package instructions. The `groups.pacman` file is used to decide which groups this client belongs to. The `packages.pacman` file is used to define what package actions the clients in each group should take.

3. In order to enact the requested package actions, `pacman` calls `stork`.

When `pacman` receives a `groups.pacman` file, it processes the file to determine which groups the client belongs to. Once `pacman` determines which groups the client belongs to, `pacman` then looks at the `packages.pacman` file to determine what package actions it should perform on the client.

The `packages.pacman` file specifies which package actions should be performed on a specific group. This makes it easy, for example, to specify that a particular package should be installed on all VMs on a physical machine, while another package should only be installed on clients used for a specific task (and thus was placed in a specified group by the administrator). Once `pacman` determines the groups the client is in, `pacman` performs the package actions specified for those groups in the `packages.pacman` file. If there are contradictory actions (such as a client is asked to install and remove the same package), the client logs an error and ignores the actions on that package.

Although `pacman` can be run manually, typically it is run by `stork_receive_update` in response to an update of the root repository metadata. A change to the root repository metadata indicates that the repository contents have changed which in turn may change the `packages.pacman` and `groups.pacman`.

2.5.4 `stork_receive_update`

The third and final component of the client tools is the `stork_receive_update` daemon. The `stork_receive_update` daemon runs on each client and keeps the root repository metadata up-to-date for each Stork repository used by the client. The root repository metadata is received from a Stork repository using both

pushing by the repository and downloading from the repository. A pushed message is received from a publish/subscribe system, PsEPR [12]. A Stork repository will push a root repository metadata with a new timestamp every two minutes even if no new data is available to notify clients that the repository is alive and PsEPR is working. This file is received by the `stork_receive_update` daemon. If `stork_receive_update` doesn't receive a root repository metadata file within a five minute period, it downloads the root repository metadata file from the repository.

Whenever a secure hash of a tarball listed in the root repository metadata file is changed, `stork_receive_update` runs `pacman`. This is because if a repository tarball has changed there is a change to the `packages.pacman`, `groups.pacman`, TP files, or package metadata. A change to any of these may mean that `pacman` must make a change to the locally installed files.

2.5.5 Bootstrapping on PlanetLab

New VMs on PlanetLab do not have the Stork client tools installed and so do not run `pacman` or `stork_receive_update` and thus cannot automatically enact package instructions. Administrators that use VMs on PlanetLab create groups of VMs in a collection called a slice. Slices are often short-lived and span many physical machines, so requiring the administrator to log in and install the Stork client tools

on every VM in a slice is impractical.

Stork makes use of a special script called the `startscript` to automatically install the Stork client tools in all of the VMs in a slice. The `startscript` is an attribute that is set on a slice and is distributed by PlanetLab's central database to the VMM. The `startscript` for a VM is run whenever the VM is instantiated. The `startscript` is also run on existing VMs whenever the attribute changes. The Stork `startscript` communicates with the nest on the physical machine and asks the nest to share the Stork client tools with it. If the nest process is not working, the `startscript` instead retrieves the relevant RPMs securely from the main Stork repository.

Some administrators also wanted the ability to easily turn off Stork updates while running experiments. To provide this functionality a `stopscript` was added. The `stopscript` stops the `stork_receive_update` daemon which prevents updates. `stork_receive_update` can be restarted by setting the slice to run the `startscript` again.

Another complication for the VMs in a PlanetLab slice is the distribution of the administrator's public key. `pacman` and `stork` require the public key of a controlling administrator for authentication. Rather than leave the key distribution to the administrator, Stork leverages PlanetLab's access control infrastructure to

distribute keys. Every PlanetLab administrator that can log into a slice must upload a SSH public key to PlanetLab Central. These keys are distributed securely by PlanetLab Central to all of the VMMs to control login for slices. Stork uses PlanetLab's SSH login keys for signing and authenticating files. This prevents an administrator from needing to manually distribute her public key.

2.6 Nest

This section briefly describes the nest to demonstrate how it fits into Stork as a whole. Since the main purpose of the nest is to share files, the majority of the description and discussion about the nest is in Chapter 4.

The Stork nest enables sharing between VMs on the same physical machine. The nest improves efficiency in package management by VMs by two mechanisms. First, the nest operates as a shared cache for VMs, allowing metadata and packages to be downloaded once and used by many VMs. Second, the nest performs package installation on behalf of the VMs, securely sharing read-only package files between multiple VMs that install the package. The nest functionality is implemented by the `stork_nest` daemon.

2.6.1 stork_nest

The `stork_nest` daemon is responsible for maintaining connections with its client VMs and processing requests that arrive over those connections (typically via a socket, although this is configurable). A client must first authenticate itself to the `stork_nest`. The authentication persists for as long as the connection is established.

Once authenticated, the daemon then fields requests for file transfer and sharing.

The nest retrieves files using the transfer functionality manager. Before a file transfer is initiated, the nest checks a cache to see if the nest has already downloaded the requested file. If so, the file is shared with the client instead of being downloaded again. Sharing files from the nest's cache is performed using the share functionality manager as described in Section 4.7.1.

The share functionality manager is also used to share packages between VMs. Packages are unpacked (but not installed) in the nest and the package files are securely shared with the requesting VMs. Unpacking the files in a package is done using the prepare functionality manager described in more detail in Section 4.7.3.

Typically, a single instance of the `stork_nest` runs on each physical machine that supports VMs and serves all of the VMs on that physical machine. However, if no nest is running or the nest process fails, the nest transfer specialized plug-in will fail to transfer files. In this case, the transfer manager will fail over to the next highest priority transfer specialized plug-in. Similarly, when installing shared packages using the package functionality manager if a failure occurs in the nest package specialized plug-in, the package will be installed without being shared.

2.7 Example Walkthroughs

Now that the different components of Stork have been explained in detail, the examples from Section 2.1 are revisited. The steps that are performed by different components are described. This is provided to clarify the interactions of components and demonstrate how the actions by individual components work toward the overall result.

The examples that were described fit into an overall process. The end goal of this process is the administrator changes the packages installed on the systems she manages. The steps are:

1. The administrator uses the GUI (Section 2.7.1) or the command-line tools (Section 2.7.2) to perform package-based configuration management. This process uploads files that contain the package instructions to the repository.

2. The repository makes the uploaded files available for download (Section 2.7.3).

3. The repository notifies the systems that updated files are available (Section 2.7.4).

4. The systems receive the notification and download the updated files (Section 2.7.5).

5. If the administrator's package instructions pertain to the system, the package instructions are enacted (Section 2.7.6).

6. Depending on the administrator's package instructions, the systems may enact different package actions including: package installation (Sections 2.7.7 and 2.7.8), removal (Section 2.7.9), or update (Section 2.7.10).

Some of the walkthroughs omit failure handling steps to avoid clutter. However, it is important to remember that there are steps that may differ if failures occur or different master configuration file options are used. For example, in some cases there will be clients that do not receive a PsEPR notification. This could occur because PsEPR failed to deliver the message or perhaps because the client is down. If PsEPR fails then `stork_receive_update` checks for updates every 5 minutes. If the client was down then when it restarts `pacman` will run. Either way `pacman` will start and obtain new root repository metadata. Regardless of how `pacman` is run, the client will perform the same steps from that point onward.

2.7.1 GUI Package-based Configuration Management

Summary: An administrator wants to update a package `foobar` on systems that are used for testing. Assume that the group `Test` already exists but that `foobar` has

not been uploaded or added to the administrator's TP file. The administrator uses the GUI to perform package-based configuration management.

1. The administrator clicks on the packages area next to the `Test` group and chooses the package action "update". Next to the package field, the administrator clicks "browse" and selects `foobar`.

2. The administrator clicks to synchronize with the repository.

3. The GUI calls `storkutil` to update the `packages.pacman` file. `storkutil` then signs the administrator's `packages.pacman` file using the administrator's private key.

4. The GUI calls `storkutil` to update the administrator's TP file to trust `foobar`. `storkutil` then signs the file using the administrator's private key.

5. The GUI authenticates with the repository using the administrator's credentials and uploads the administrator's `packages.pacman` file, TP file, and `foobar`.

2.7.2 Command Line Package-based Configuration Management

Summary: An administrator wants to deploy code automatically as part of a build process. The administrator has already assigned clients to a group `Test` and wants

the package she builds added to the clients in the `Test` group.

An example of how this may be done is in Figure 2.7. This script creates a package with the administrator's code. The package is added to the TP file and the `packages.pacman` file is changed so that the clients will update the package. The files are then uploaded to the repository. This is identical in end result to the walkthrough in the previous section in which the administrator used the GUI.

2.7.3 Repository Uploads

Summary: An administrator uploads files to a repository. In this walkthrough an administrator uploads `foobar.rpm`, a TP file, and a `packages.pacman` file to the repository. Each file upload is treated as a separate action by the repository.

1. The administrator authenticates with the repository.
2. The administrator uploads `foobar.rpm` to the Stork repository.
3. The repository places `foobar.rpm` in a directory named by the secure hash of the package contents. `foobar.rpm` is now available for download from that location.
4. The repository extracts the package metadata from `foobar.rpm` and stores it in a file whose name is the secure hash of the contents.

```
# Build and package the code into foobar.rpm
...

# Add the package to the administrators TP file.
storkutil addfile foobar.rpm

# Instruct the clients in the Test group to update the package
storkutil pacpackages group Test update foobar.rpm

# upload the package, TP file, and packages.pacman file
...
```

FIGURE 2.7. **Example build script using `storkutil`.** This script builds a package containing an administrator's program, adds the package to the administrator's TP file, and instructs the `Test` group to update the package. The upload can be done using `curl` or other tools but is not shown for brevity.

5. The repository updates the package metadata tarball to include the new package metadata file that was extracted from `foobar.rpm`.

6. The repository updates the root repository metadata to have the new file size and secure hash of the package metadata tarball. The new root repository metadata is signed using the repository's private key.

7. The repository pushes out the new root repository metadata via PsEPR.

The administrator now uploads the TP file. There is no need to re-authenticate because the session is still active.

1. The administrator uploads the TP file to the Stork repository.

2. The signature on the TP file is verified using the administrator's public key that is embedded in the file name. The new file replaces the existing file if the signature is valid and the timestamp is newer.

3. The repository updates the tarball on the repository that contains the TP files to include the new TP file.

4. The repository updates the root repository metadata to include the new file size and secure hash of the TP tarball. The new root repository metadata is signed using the repository's private key.

5. The repository pushes out the new root repository metadata via PsEPR.

The handling of the `packages.pacman` file is similar to the handling of the TP file except that the `packages.pacman` file is added to the pacman tarball on the repository instead of the TP tarball.

2.7.4 Pushing Repository Notifications

Summary: A repository has new content and pushes a notification to clients.

1. A Stork repository has updated content and as a result has created a new root repository metadata file. The repository signs the root repository metadata.
2. The repository uses the publish/subscribe system PsEPR to push out the new root repository metadata to the clients.
3. The clients are running `stork_receive_update` and therefore obtain the new root repository metadata pushed out over PsEPR.
4. The `stork_receive_update` daemon wakes up the `pacman` daemon.

`pacman` is now ready to download the updated repository metadata.

2.7.5 Selective Content Updates

Summary: A client receives a PsEPR notification from the repository and downloads only the changed metadata tarballs. This walkthrough assumes that the TP file, pacman, and package metadata tarballs have changed and the master configuration file tarball is the same as the version on the client.

1. `stork_receive_update` receives a new root repository metadata file from a PsEPR update and provides this to `pacman`. `pacman` calls the transfer functionality manager to synchronize the directory containing the repository metadata tarballs with the local copy.

2. To do the synchronization, the transfer functionality manager checks the secure hashes of the copies of the local tarballs for this repository.

3. The transfer functionality manager compares the hashes of the local tarballs to the secure hashes listed in the root repository metadata.

4. The tarballs containing the TP files, pacman files, and package metadata have different secure hashes than are listed in the root repository metadata. These tarballs are downloaded using the preferred transfer specialized plug-in.

The metadata from the repository is now synchronized on the client. `pacman` processes the files and enacts any new package instructions.

2.7.6 Enacting Package-based Configuration Management Instructions

Summary: `pacman` is run on a client with a `packages.pacman` file that instructs the `Test` group to install a package `foobar`.

It is important to note that this action is performed by every client the administrator manages because this is the step in which a client determines if the package instructions apply to it. Different clients will perform different actions depending on what groups the client is in. Additionally, a client that reboots or is newly created will run `pacman` when starting and at that time will enact actions for the groups the client is in.

1. `pacman` parses the `groups.pacman` file to determine which groups the client is in.

2. `pacman` enacts the pending package actions for this client. If this client is in the `Test` group, `pacman` invokes `stork` to update `foobar`.

2.7.7 Installing a Package

Summary: `stork` is run to install `foobar`. Suppose that `foobar` is a RPM without any dependencies.

1. `stork` uses the transfer functionality manager to synchronize the directories containing the metadata tarballs on the repositories the client uses. Since `pacman` also does this, having `stork` perform this action may download the root repository metadata unnecessarily, but is necessary when `stork` is run manually or multiple updates happen in a short time period.

2. The package functionality manager verifies that the specified version of `foobar` is not already installed. If `foobar` is installed, `stork` prints a message and exits.

3. `stork` calls the package functionality manager to search the package metadata for `foobar`. If no candidate is found then `stork` exits and logs an error message that the package cannot be found. Multiple candidates may be returned if the package metadata contains several versions of `foobar`, however in this example a single version of `foobar` is returned.

4. `stork` verifies that the administrator trusts the candidate version of `foobar` (Chapter 3).

5. The package functionality manager checks and discovers that `foobar` does not depend on other packages.

6. `Stork` calls the transfer functionality manager to retrieve `foobar`.

7. `Stork` now has a local copy of `foobar`. The package functionality manager

queries the package specialized plug-ins to find one that can install the package. The `stork_rpm` specialized plug-in is selected to install RPM packages. Because `foobar` is a RPM package, `stork_rpm` can process it. The package manager builds a transaction list and passes it to `stork_rpm`, instructing it to install `foobar`.

8. The `stork_rpm` specialized plug-in installs `foobar`.

2.7.8 Multiple Package Types and Dependencies

Summary: A Stork administrator instructs `stork` to install `foobar`. `foobar` depends on `glibc` which is installed. `foobar` also has a dependency on `xyz.tar`, however, `xyz.tar` is not installed.

1. `stork` uses the transfer functionality manager to synchronize the directories containing the metadata tarballs on the repositories the client uses.

2. The package functionality manager verifies that `foobar` is not already installed by asking the package specialized plug-in.

3. The package functionality manager searches the package metadata and finds the metadata for `foobar`.

4. `stork` verifies that the administrator trusts `foobar` (described in Chapter 3).

5. `stork` now has the candidate `foobar`. However, `foobar` depends on `glibc` and `xyz.tar`.

6. The package functionality manager discovers that `glibc` is already installed by asking the package specialized plug-in. The `glibc` dependency is resolved and so need not be considered further.

7. The package functionality manager discovers that `xyz.tar` is not installed by asking the package specialized plug-in.

8. The package functionality manager searches the package metadata and finds the metadata for `xyz.tar`.

9. `stork` verifies that the administrator trusts `xyz.tar` (described in Chapter 3).

10. `stork` now has the candidates `foobar` and `xyz.tar`. `xyz.tar` has no dependencies so there are no unresolved dependencies.

11. `Stork` calls the transfer functionality manager to retrieve `foobar` and `xyz.tar`.

12. There are now local copies of `foobar` and `xyz.tar`. Because `foobar` is dependent on `xyz.tar`, `xyz.tar` must be installed first. The package functionality manager queries the package specialized plug-ins and determines that the package specialized plug-in that handles tarballs (`stork_tar`) can install `xyz.tar`.

13. `stork_tar` is instructed to install `xyz.tar` by the package functionality manager and does so.

14. The package functionality manager queries the package specialized plug-ins and determines that `stork_rpm` can install `foobar`.

15. `stork_rpm` is instructed to install `foobar` by the package functionality manager and does so.

2.7.9 Removing a Package

Summary: `stork` is run to remove `foobar`.

1. The package functionality manager queries the `stork_rpm` specialized plug-in to see if `foobar` is installed. If `foobar` is not installed, `stork` prints a message and exits.

2. The package functionality manager checks to see if any installed packages depend on `foobar`. The package functionality manager does this by querying the `stork_rpm` specialized plug-in. If packages depend on `foobar`, `stork` prints a message and exits.

3. The package functionality manager builds a transaction list and passes the transaction list to `stork_rpm`, instructing `stork_rpm` to remove `foobar`.

4. The `stork_rpm` specialized plug-in removes `foobar`.

2.7.10 Updating a Package

Summary: `stork` is run to update `foobar`. Suppose that `foobar` is a RPM without any dependencies and that an existing, out-of-date copy of `foobar` is installed.

1. `stork` uses the transfer functionality manager to retrieve the current metadata tarballs from the repositories it uses.

2. The package functionality manager verifies that the preferred version of `foobar` is not already installed. If the preferred version of `foobar` is installed, `stork` prints a message and exits.

3. `stork` calls the package functionality manager to search the package metadata for `foobar`. If no candidate is found then `stork` exits and logs an error message that the package cannot be found. Multiple candidates may be returned if the package metadata contains several versions of `foobar`, however in this case a single version of `foobar` is returned.

4. `stork` verifies that the administrator trusts the candidate version of `foobar` (Chapter 3).

5. `stork` now has a candidate for `foobar`. The package functionality manager checks and discovers that `foobar` does not depend on other packages.

6. `Stork` calls the transfer functionality manager to retrieve `foobar`.

7. `Stork` now has a local copy of `foobar`. The package functionality manager queries the package specialized plug-ins to find one that can install the package. The `stork_rpm` specialized plug-in is selected to install RPM packages. `Stork` builds a transaction list and passes the list to `stork_rpm`, instructing `stork_rpm` to remove the existing `foobar` and install the preferred version of `foobar`.

8. The `stork_rpm` specialized plug-in removes the existing version of `foobar` and installs the new version of `foobar`.

2.8 Related Work

This section describes related work that manages software on clients. In particular, package managers (Section 2.8.1), software installers (Section 2.8.2), software update systems (Section 2.8.3), and configuration management systems (Section 2.8.4) are discussed.

A package manager [56] serves as a central mechanism to manage the software on a client. Package managers use a standard format and normally use a single installation database for all installed software.

In contrast, a software installer [73] is bundled separately for each piece of software. Each software installer may have a different format for software installed on the same client. Each software installer usually manages installations separately.

Unlike package managers and software installers that may be used to install and remove software, a software update system is used by a piece of software to update the version of the software that is running on a client to the latest version. A software update system does not install a version when one does not exist or remove a piece of software.

The focus of a configuration management system is to change the configuration of clients that are running software. Most configuration management systems are

geared toward creating custom configuration files and other low level operations rather than changing the packages installed on a client.

Related work that is related to security or sharing is discussed in Sections 3.10 and 4.10 respectively.

2.8.1 Package Managers

Most package managers are comprised of two components called a *package installer* and a *dependency resolver*. The package installer manages the installation of the files in a package on a client, and typically keeps a database to track the packages that are installed. There is usually one package installer per client. The package installer is usually standardized for the distribution; RPM [68] and DPKG [26] are examples of package installers.

A dependency resolver handles communication with repositories as well as retrieving packages that a package needs to operate (this process is called *dependency resolution*). Dependency resolvers use a package installer to perform the actual package installation, removal, or update. Administrators usually interact with a dependency resolver rather than invoking a package installer directly. While dependency resolvers and package installers work together to form a package manager, it is common to refer to a package manager using only the dependency

resolver's name. Example dependency resolvers include APT [5], YUM [92], and `stork`.

2.8.1.1 Package Installers Two popular package installers are RPM and DPKG. Instead of using RPM or DPKG, many dependency resolvers operate on tarball based formats and provide limited package installer functionality themselves.

RPM is a package installer that operates on packages in the RPM format. The RPM package installer is a command line tool that is "capable of installing, uninstalling, verifying, querying, and updating computer software packages" [68]. The RPM package installer is used by many dependency resolvers including YUM [92], YaST [90], APT-RPM [6], Urpmi [81], and `stork` [76].

DPKG [26] operates on packages in the DEB format. Similar to the RPM package installer, DPKG installs, removes, updates, and queries packages. The DPKG and RPM package installers are similar, the major difference being how the packages are built [54, 55]. The DPKG package installer is used by the dependency resolver APT [5].

Many of the other package installers manage packages that are essentially just tarballs with an install script and some package metadata. Some examples of this are ebuild [29] (which is used by emerge [63]) and the tarball support for `stork` [76],

Pacman [7], and Slaktool [72].

2.8.1.2 Dependency Resolvers There are many dependency resolvers including APT [5], APT-RPM [6], emerge [63], Pacman [7], pkg_add [13], Slaktool [72], `stork` [76], Urpmi [81], YaST [90], and Yum [92]. Most dependency resolvers are very similar in terms of the functionality they provide. All of the dependency resolvers retrieve, configure, and install packages on a client and resolve dependencies. Many of the dependency resolvers have GUI-based front ends in addition to a command line tool. The GUI-based front ends access the same underlying functionality as the dependency resolver but present a different interface to the administrator.

APT was one of the first dependency resolvers and was created to use the package installer DPKG. However, APT has also been ported to use other package installers, including RPM (where it is named APT-RPM).

The emerge dependency resolver is part of a package manager that is explicitly called Portage [63]. Portage is a framework that is meant to support multiple architectures and package formats across different distributions. There are implementations of Portage for Mac OS X, FreeBSD, and Solaris.

Pacman is a dependency resolver primarily used in Arch Linux. Pacman uses zipped

tarballs as packages.

The pkg_add dependency resolver (commonly called Ports) is used on BSD clients to manage packages. One of the main aims of Ports is to be portable, and Ports is used on a variety of different operating systems.

Slaktool is the dependency resolver used in Slackware. Slackware distributes packages in source format and compiles software on the client as part of the installation process.

The `stork` dependency resolver is discussed in this dissertation. `stork` is unique for its security and sharing functionality (as are described in Chapters 3 and 4). Another uncommon characteristic of `stork` is that `stork` supports multiple package installers (tarball and RPM) on the same client.

Urpmi is a dependency resolver used in Mandriva Linux. Urpmi uses RPM to perform package installation.

YaST is the dependency resolver for SuSE distributions. YaST is a popular, graphical utility for package management. YaST uses RPM to perform package actions.

The YUM dependency resolver is used on many distributions including RedHat, Fedora, and CentOS. YUM was developed at Duke and uses RPM to perform

package actions.

Dependency resolvers are discussed in more detail in Chapter 3 with APT and YUM used as examples of different security philosophies for package management.

2.8.2 Software Installers

In contrast to a package installer that installs each piece of software on a client, a software installer is tailored to the particular software component it installs. Software installers are most popular on Windows and Mac OS X systems.

There are many software installers including InstallShield [41], Wise [88], and StuffIt [77]. A software installer maintains its own record of the files that are installed. In many cases software installers also include an uninstall program for the purpose of removing the software. Software installers provide similar functionality to package managers in that they install software on a client. However, software installers do not typically perform dependency resolution. As a result, many software installers recommend bundling all of the necessary software with the installer.

2.8.3 Software Update Systems

A software update system is used to update an installed piece of software to the latest version. Software update systems do not support the installation or removal of the software, only update. Software update systems are in most cases run automatically by the software to check to see if a newer version of the software is available (often when the software starts). Software update systems are often created specifically for a piece software instead of being reused for multiple pieces of software. Popular programs that use software update systems are Mozilla Firefox [32], Adobe Acrobat [1], Microsoft Office [50], Skype [71], and Adium [2].

A few software update systems update multiple different software applications distributed by the same vendor, examples of which are WindowsUpdate [35, 86] and the Apple software updater [47].

Software update systems do not support dependency resolution and are only useful when the software they update has already been installed. This limits the applicability of software update systems.

2.8.4 Configuration Management Systems

There are many configuration management systems that focus on configuring a client. The goal of configuration management systems differs from tool to tool, but the commonality is that the administrator uses a configuration management system to change the configuration of the software on a client. This often involves editing configuration files, starting and stopping software, or monitoring a running software program. Configuration management systems typically rely on package managers (or similar functionality) to install the software that will be configured. Some examples of configuration management systems are Cfengine [18], SmartFrog [37], Red Hat Network [67], GrADS [10], Plush [3], and AppManager [4].

While Stork provides some rudimentary package-based configuration management, other configuration management systems provide much more control over the configuration of a client. Stork's advantage over other configuration management systems is that the configuration management in Stork is tailored to package management and so does not present the administrator with functionality that is not necessary for package-based configuration management. Of course, the drawback of Stork's approach is that the administrator cannot do more than manage packages without using another configuration management system.

2.9 Conclusion

This chapter provided a description of the components that comprise Stork. The package-based configuration management tools are used by an administrator to manage packages on clients. The repository serves administrator uploaded packages and metadata to clients. The client tools enact the administrator's package instructions and perform package management. The nest shares files between VMs on the same physical machine to improve the efficiency of package management. Details describing how security and sharing are provided are deferred to Chapters 3 and 4 where they are explored in more detail.

Chapter 3

Security

Package managers provide a privileged, central mechanism for the management of software on a client. Since package managers update software on a client, package management security is essential to the overall security of the client. Package managers provide security by verifying signatures on data that is downloaded from a repository. Data is signed by an *entity*. An entity possesses a private key and public key and makes decisions about what packages or metadata should be trusted. Administrators, developers, projects, repositories, and distributions are all examples of entities because they may sign packages and package metadata to help clients authenticate the data the client downloads.

Despite signatures added by entities, there are many ways for an attacker to circumvent security in package managers. Nine attacks on package managers are described and shown to be feasible on the popular package managers APT [5] and YUM [92]. These attacks may be used to install malicious packages on clients, prevent clients from getting security updates, and exhaust resources on clients

(commonly causing mail delivery to stop, databases to be corrupted, and logging to fail). Unfortunately, fixing many of these problems requires fundamental changes in the security architectures of existing package managers.

The nine attacks are examined and three common themes can be drawn.

1. Package managers should not trust the repository. This means that both the data retrieved from a repository and the protocol used when communicating with a repository must be scrutinized by the package manager. Failure to scrutinize repository interactions allows a man-in-the-middle attacker or attacker that controls a repository to wreak havoc on a client whose package manager uses the repository.

2. The entity with the most information should be the one that signs. This means that an entity should not sign a package simply because the package was previously signed by another entity. Since the entity that builds a package has the most information about the package, that entity should be the one that signs to certify that the package metadata is valid. This argues that the developer or developers that maintain the package should sign the package, not the distributor.

3. The package manager must prevent key misuse to prevent the installation of untrusted packages. Thus, the package manager needs an effective way of

determining if a key is revoked. Preventing key misuse also means that an entity must be able to restrict the trust given to another entity. For example, if an entity is trusted to sign packages for a specific project, the entity should not be automatically trusted to sign packages for another project.

The underlying themes of these attacks describe in what areas of package managers must be defended in order to retain security. However, these themes do not describe how the necessary security mechanisms should be provided. To that end, three design principles for security in package management were developed for Stork.

1. Using selective trust delegation, administrators can trust an entity (such as a developer) to know the validity of a limited subset of packages. In addition, selective trust delegation can be used to prevent exposing the project entity's private key to individual developers. Selective trust delegation also provides a natural mechanism for key revocation.

2. Having customized repository views means that each client "sees" a different repository, which is actually an amalgamation of the trusted packages on all repositories the client is using. Customized repository views prevent malicious entities from compromising the security of clients that do not trust the entity.

3. If the repository is treated as an untrusted entity, then even a root and private key compromise of a package repository does not compromise the security of

the repository's clients. This is because the client's package manager scrutinizes the interactions with the repository.

This chapter presents the security architecture of Stork that follows these design principles. Besides the mechanisms used to directly support these design principles, dependency resolution is also discussed as it mitigates certain kinds of attacks. Stork is then analyzed and shown not to be vulnerable to the attacks that impact other package managers.

3.1 Map

This chapter explores package management security by first examining attacks that package managers may be vulnerable to and the common themes behind these attacks (Section 3.2). Then background information is provided about existing package managers to understand why the package managers are vulnerable to attack (Section 3.3). Section 3.4 discusses the feasibility of an attacker gaining the necessary requirements to launch the attacks described. Section 3.5 describes the result of these attacks on existing package managers. Section 3.6 discusses how existing package managers can prevent these attacks. Unfortunately many of these attacks exploit flaws in the security architecture of existing package managers and so cannot be easily fixed. Principles for designing a security architecture that protects against these attacks are then discussed (Section 3.7). Then the implementation of this security architecture in Stork is described (Section 3.8). The attacks in Section 3.2 are revisited in Section 3.9 to illustrate how Stork prevents the attacks from being effective. Section 3.10 discusses related work and Section 3.11 concludes.

3.2 Attacks

This section describes nine attacks to which package managers may be vulnerable. Then these attacks are examined and underlying themes are discussed. A summary of the attacks and the themes can be found in Table 3.1.

3.2.1 Attack Descriptions

The nine attacks are as follows:

1. *Slow Retrieval:* In this attack, a package manager tries to download a file from a repository. An attacker slows the retrieval so that the communication lasts for many hours or even days. The attacker does this by sending back a few bytes of data a minute — just enough to keep the connection open. The goal is to make the package manager "pause" so the client will not contact other repositories to retrieve package updates.

2. *Endless Data:* In this attack, an attacker returns an infinite stream of data in response to any request. The goal is to make the client's package manager continue to download data and exhaust resources such as disk space and network bandwidth.

Attack Name	Description	Theme
Slow Retrieval	An attacker slows repository communication so that package managers will "pause" and will not contact other repositories to retrieve package updates.	(1)
Endless Data	An attacker that controls a repository (or man-in-the-middle attacker) returns an endless stream of data in response to any file request.	(1)
Replay Old Metadata	An attacker provides old metadata that is correctly signed.	(1)
Extraneous Dependency	An attacker changes the metadata for a package to indicate it depends on a package or packages of the attackers choice.	(2)
Depends on Everything	An attacker changes the metadata for a package to indicate it depends on everything.	(2)
Unsatisfiable Dependencies	An attacker causes a package manager to ignore valid packages because forged metadata indicates unsatisfiable dependencies.	(2)
Provides Everything	An attacker changes the metadata for a package to indicate it provides any dependency the administrator requests.	(2)
Use Revoked Keys	An attacker uses a revoked key to get administrators to install packages	(3)
Escalation of Privilege	An attacker compromises a key trusted for signing a specific group of packages and then gets administrators to accept signed malicious versions of other packages.	(3)

TABLE 3.1. This table lists attacks, with a description and the theme of the attack. The security themes are numbered (1) Don't trust the repository (2) The trusted entity with the most information should be the one that signs (3) Prevent key misuse to prevent the installation of untrusted packages.

3. *Replay Old Metadata:* An attacker records old metadata from a repository. When a package manager requests metadata from the repository the attacker returns the old metadata. The attacker's goal may be to have the package manager install outdated packages with security vulnerabilities.

4. *Extraneous Dependency:* An attacker changes the package metadata for a target package to include an additional dependency. The attacker may use an extraneous dependency attack to have a package manager that installs the target package also install a package with a security vulnerability.

5. *Depends on Everything:* An attacker changes the package metadata to indicate that a package depends on many other packages. This will cause a package manager to download many packages to resolve the dependency. The depends on everything attack wastes the client's network bandwidth and disk space.

6. *Unsatisfiable Dependencies:* An attacker changes the package metadata for a package so that the target package depends on a package that does not exist. This prevents the package manager from installing the target package.

7. *Provides Everything:* An attacker changes the package metadata for a package to indicate that the package provides every dependency. This means that the package will be considered to fulfill any dependency of another package. An attacker may use this to have a package installed more frequently.

8. *Use Revoked Keys:* An attacker that has compromised an entity's key signs packages and / or metadata so that clients install malicious packages. Another entity that has the authority to revoke the key is attempting to do so. The attacker is trying to have clients accept the malicious information before the clients notice that the key has been revoked.

9. *Escalation of Privilege:* An attacker compromises the key of an entity trusted to sign a specific group of packages. The attacker modifies a package that is not in that group and then uses the entity's key to sign the package. The attacker provides the package to the package manager on a client to see if the package manager accepts a package signed by an incorrect (but trusted) entity.

3.2.2 Attack Themes

The previous section described attacks on package managers. This section draws correlations between vulnerabilities. If there are commonalities between programming and design errors in package managers, then perhaps developers can avoid these issues. The attacks described throughout this section (depicted in Table 3.1) have three common themes:

1. Package managers should not trust the repository. Package managers should scrutinize both the data retrieved from a repository and the communication

with the repository. The slow retrieval and endless data attacks would work if the package manager does not scrutinize the communication with the repository. The replay old metadata attack would be effective if the package manager does not check that the metadata returned from a repository is current.

2. The entity with the most information should be the one that signs. This means that an entity should not sign a package simply because the package was previously signed by another entity. For example, many repositories re-sign all of the package metadata provided by developers. One problem with this is that all of the package metadata will be signed by a single entity's key, which presents problems when a package should no longer be trusted. Since the repository entity signed the package, the repository entity's key must be revoked. Since the repository entity was trusted for all of the package metadata, if the repository entity's key is revoked then *all* packages should not be trusted.

   In addition, if the entity with the most information is the one that signs, the entity that builds a package should sign the package metadata. This is true because the entity that builds the package has the most information about the validity of the package. The extraneous dependency, depends on everything, unsatisfiable dependencies, and provides everything attacks are effective if the

package metadata can be modified by an attacker. An attacker can modify the package metadata but not the package when the package metadata is unprotected or the package metadata is signed by an entity other than the developer and that entity's key is compromised. If the package metadata is signed by the developer, then the only way the package metadata can be altered is with the developer's key.

3. The package manager must prevent key misuse to prevent the installation of untrusted packages. This means that the package manager needs an effective way of determining if an entity's key is revoked. The use revoked keys attack exploits the difficulty of revoking an entity's key before the entity's key is used by the attacker. Preventing key misuse also means that there must be able to be restrictions on the trust given to an entity. For example, if an entity is trusted to sign packages from a specific project, the entity should not be automatically trusted to sign packages for any other project. Failure to have selectivity in the amount of trust given to an entity can lead to escalation of privilege attacks.

3.3 Background

The previous section presented attacks against package managers and the underlying themes of those attacks. This section provides background information on existing package managers that is necessary to understand existing package managers' vulnerability to attack.

Most package managers can be split into two basic components: a *package installer* and a *dependency resolver*. A package installer is a low-level component that removes, installs, and updates the packages on a client. Example package installers are RPM [68] and DPKG [26]. A dependency resolver is a high level component that handles communication with repositories as well as dependency resolution. Example dependency resolvers include `stork` [76], APT [5], and YUM [92]. Since dependency resolvers and package installers work together to form a package manager, it is common to refer to a package manager using only the dependency resolver's name. Since most dependency resolvers only support a single package installer (with Stork's support of tarballs and RPMs a notable exception), it is clear from referring to the dependency resolver which package installer is meant.

Before the characteristics of package managers are looked at in detail, this section provides some statistics on which package managers are popular. Popularity is important because the more popular software is, the more clients will be

compromised if a vulnerability exists. This chapter focuses on popular package managers with diverse characteristics. Other package managers are categorized in Section 3.10 based upon similarity to the popular package managers.

3.3.1 Package Manager Popularity

Since package installers and dependency resolvers are tied to a specific distribution of Linux, the popularity of distributions has a high correlation with package installer and dependency resolver popularity. According to DistroWatch [25] and Netcraft [52] (shown in Table 3.2), DEB and RPM file formats are overwhelmingly the most popular. As a result the corresponding package installers DPKG and RPM are also popular. As for dependency resolvers, on the desktop APT/APT-RPM (45.2%) is clearly the most popular, with YUM (6.2%), emerge (6.9%), and YaST (10.7%) all having significant market share. For servers, YUM (52%) is the clear leader with APT (25%) also having strong market share.

APT/DPKG and YUM/RPM are the most popular package managers and, as will be discussed later, APT and YUM use different philosophies to provide package management security. Other package managers tend to be "APT-like" or "YUM-like" in their security model. This majority of this chapter focuses on APT/DPKG and YUM/RPM and later in Section 3.10 package managers are

categorized as like either APT or YUM.

3.3.2 Package Formats

Package formats consist of an archive containing files and, in the case of RPM and DEB, additional metadata. For a given RPM or DEB package, the additional metadata contains information about the other packages the package needs to operate (the package's dependencies), functionality the package possesses (what dependencies the package provides), and various other information about the package itself.

In RPM there is space for only a single signature. DEB packages have no standard field for signatures (and DEB files are not usually signed) but extensions exist to support signatures [24, 27].

3.3.3 Package Installers

The package installer is normally tied to a specific package format. The role of the package installer is to unpack the files from the package to the appropriate location, track installed packages, and install, remove, or update packages. The package installer keeps a database that records which packages are installed. Most package

Distribution	Desktop Usage	Server Usage	Package Format	Package Installer	Dependency Resolver
Red Hat Enterprise	0.9%	34%	RPM	RPM	YUM
Debian GNU/Linux	8.9%	25%	DEB	DPKG	APT
Ubuntu	24.1%	N/A	DEB	DPKG	APT
Fedora	5.0%	16%	RPM	RPM	YUM
SUSE Linux Enterprise	0.8%	11%	RPM	RPM	YaST
openSUSE	9.9%	N/A	RPM	RPM	YaST
Gentoo Linux	4.5%	2%	SRC	ebuild	emerge
PCLinuxOS	5.6%	N/A	RPM	RPM	APT-RPM
Mandriva Linux	2.6%	2%	RPM	RPM	urpmi
CentOS	1.2%	2%	RPM	RPM	YUM
Linux Mint	2.5%	N/A	DEB	DPKG	APT
Sabayon Linux	2.4%	N/A	SRC	ebuild	emerge
Slackware	2.2%	N/A	TGZ		slaktool
KNOPPIX	2.2%	N/A	DEB	DPKG	APT
Arch Linux	1.9%	N/A	TGZ		pacman
MEPIS Linux	1.9%	N/A	DEB	DPKG	APT

TABLE 3.2. The usage information shows the average popularity of desktop Linux distributions from multiple sources as reported by DistroWatch [25]. The server numbers are reported from Netcraft [52] (Cobalt numbers are omitted because the distribution is not listed). Distributions without a package installer listed typically use `tar` to unpack files but do not provide the functionality found in package installers.

installers also provide a signature verification mechanism to detect package tampering or corruption. There is typically only one package installer per client and the package installer is normally standardized for the distribution.

RPM [68] is both the name for a package installer and the format of the package files that the package installer uses. The package installer is command line driven and is "capable of installing, uninstalling, verifying, querying, and updating computer software packages" [68]. While RPM is sometimes used standalone, it is common to run a dependency resolver like APT-RPM or YUM on top of RPM.

DEB packages are used by a package installer called DPKG [26]. DPKG performs similar actions as the RPM package installer, allowing the installation, uninstallation, and querying of computer software packages. There are subtle differences between DPKG and RPM [54], but the differences are interesting primarily from a functionality and ease of use standpoint rather than a security standpoint.

3.3.4 Dependency Resolvers

The dependency resolver gathers information about packages available on package repositories. Almost all dependency resolvers automatically download requested packages as well as any additional packages that are needed to resolve the

dependencies of the requested packages (hence the name "dependency resolver"). For example, a requested package `foo` may depend on `libc` and `bar`. If `libc` is already installed, then `libc` is a dependency that has been resolved (so no package must be added for this dependency). If there is no installed package that provides `bar`, then `bar` is an *unresolved dependency* and a package that *provides* `bar` must be installed before `foo` may be installed. The dependency resolver may be able to locate a package that provides `bar` on a repository. Note that this does not need to be a package with the name `bar`. A package may provide *virtual dependencies* of another name. This is useful if a package needs a program that provides some functionality but it doesn't matter what the program is (such as an email client or web browser).

A package that is chosen to fulfill dependencies may have unresolved dependencies of its own. Packages are continually added to the list of packages to be installed until either the package manager cannot resolve a dependency (and produces an error) or all dependencies are resolved. Dependency resolution is described in more detail in Section 3.8.5.

The actual installation is done by the dependency resolver calling the package installer. It is common for multiple dependency resolvers [6, 76, 81, 92] to support a single lower-level package installer (RPM). It is possible (but uncommon) for different administrators that use the same distribution to use different dependency

resolvers.

One popular dependency resolver is APT. APT is a dependency resolver originally created to use the package installer DPKG in Debian. However, APT has also been ported to use other package installers, including RPM [6] (where it is named APT-RPM). APT automates the retrieval, configuration, and installation of packages by resolving dependencies. APT also handles communication with repositories. To address security concerns with APT (particularly with signatures), the secure-APT project [70] was created. Recent versions of APT use the changes added for secure-APT. This work focuses on the newer versions of APT with the secure-APT changes and ignores older versions since older versions are known to be insecure.

Another popular dependency resolver is YUM [92]. YUM is used to install and remove packages on clients with RPM package installers. YUM performs automatic dependency resolution and repository communication.

YUM and APT both include auto-update mechanisms. When enabled, these auto-update mechanisms update older versions of installed packages with newer versions of the packages as newer versions become available, typically within 1 day of the new packages being added.

3.3.5 Package Repository

Package repositories are usually just web servers used to provide packages and package metadata. The package metadata is the metadata in the package format that is extracted and stored separately. Often the package metadata for all packages on a repository are put into a single tarball.

Package repositories store packages, package metadata, and the *root repository metadata* file. The root repository metadata file is called different things in APT (`Release`) and YUM (`repomd.xml`) but the contents are similar. The root repository metadata provides the location and secure hashes of the tarball that contains the package metadata and may also refer to the location of the packages.

It is common for a distribution or large organization to have multiple repositories that host the same data for redundancy and load balancing purposes. These additional repositories are called *mirrors*. Mirrors typically contain exactly the same content as the main repository and are updated via `rsync` or a similar tool.

3.3.6 Security Philosophy

APT and YUM use different techniques to provide security. APT focuses on securing the repository metadata rather than signing packages. An APT repository

optionally provides a signature for the `Release` file (root repository metadata) in a file called `Release.gpg`. This allows APT to verify that the `Release` file is signed by the repository key and therefore came from the repository. The `Release` file contains the secure hashes of the package metadata on the repository. The package metadata contains secure hashes of the packages themselves. The packages are not usually signed.

Instead of signing the repository metadata, YUM uses signatures on packages to provide security. A YUM distribution maintainer signs all of the packages on the repository. YUM verifies package signatures after downloading packages. YUM does not sign package metadata.

To verify the identity of a repository or mirror, YUM uses HTTPS in Red Hat Enterprise Linux. Other distributions of APT and YUM do not support HTTPS on their publicly available repositories and mirrors.

3.4 Attack Feasibility

This section examines the attacker's requirements to launch the attacks described in Section 3.2 on APT and YUM. There are two basic questions this section tries to answer. First, what resources does an attacker require to launch an attack? Second, is it possible for attackers to obtain the necessary resources to launch these attacks?

There are increasing requirements of resources that allow a greater number of attacks (Table 3.3). The first requirement is that the attacker must be able to impersonate a repository to launch a basic attack (Section 3.4.1). Once the attacker is able to impersonate a repository, the second requirement is to sign metadata, allowing more severe attacks (Section 3.4.2). If an attacker is able to impersonate a repository and sign metadata, then the third requirement is to sign packages, allowing the most severe attacks (Section 3.4.3).

An alternative avenue of attack is to compromise a developer key, allowing the attacker to submit arbitrary packages to the repository through the legitimate update channels without the need to compromise the repository itself (Section 3.4.4). This does not require any compromise other than a developer key and allows the launch of any attack.

The feasibility of obtaining the requirements to launch these attacks is now

Attack Name	Description	Requirement
Slow Retrieval	An attacker slows repository communication so that package managers will "pause" and will not contact other repositories to retrieve package updates.	Repository
Endless Data	An attacker that controls a repository (or MITM attacker) returns an endless stream of data in response to any file request.	Repository
Replay Old Metadata	An attacker provides old metadata that is correctly signed (perhaps to prevent new packages from being considered)	Repository
Extraneous Dependency	An attacker changes the metadata for a package to indicate it depends on a package or packages of the attackers choice.	Repository Key
Depends on Everything	An attacker changes the metadata for a package to indicate it depends on everything.	Repository Key
Unsatisfiable Dependencies	An attacker causes a package manager to ignore valid packages because forged metadata indicates unsatisfiable dependencies	Repository Key
Provides Everything	An attacker changes the metadata for a package to indicate it provides any dependency the client requests.	Repository Key
Use Revoked Keys	An attacker uses a revoked key to get clients to install packages	Revoked Key
Escalation of Privilege	An attacker compromises a key trusted for signing a specific group of packages and then gets clients to accept signed malicious versions of other packages.	Package Key

TABLE 3.3. This table lists attacks, a short description of the attack, and what the attacker requires to launch an attack. The first three attacks require only the ability to impersonate a repository. The next four attacks require both the ability to impersonate a repository as well as the compromise the entity's key that is used to sign repository metadata. The last two attacks require the ability to impersonate a repository, the repository metadata key, and an entity's key trusted to sign packages.

described.

3.4.1 Impersonate a Repository

The attacks against package managers require an attacker to provide traffic on the behalf of a repository. There are three ways this can be done: man-in-the-middle (MITM) attacks, control of a repository, or control of a mirror.

In MITM attacks, the attacker intercepts traffic between two communicating parties and provides its own data instead. These attacks are relatively simple to launch [36, 91] and there are toolkits available that automate the process [28, 80]. When an insecure protocol such as HTTP is used, a MITM attacker is equivalent to an attacker that can control a repository through direct compromise. Using a secure protocol such as HTTPS can prevent MITM attackers from masquerading as a repository, however with the exception of Red Hat Enterprise Linux, HTTPS is not widely used.

A repository is typically very similar to a web server, and attacks that compromise web servers, such as software exploits, may allow the attacker to modify content on the repository. Given the success of worms written to exploit bugs in Linux [46, 66], a direct, exploit-based compromise of the repository is also a possibility.

Third-party developers often set up their own repositories to provide software that is not included in the distribution. An attacker could exploit this common practice by setting up its own repository purporting to provide third-party software and try to lure administrators to use the repository. However this only affects the subset of administrators that use this software.

Another avenue of attack is for an attacker to obtain control of a mirror. To evaluate the feasibility of controlling a mirror of a popular distribution, YUM and APT mirrors were set up. A fictitious company (Lockdown Hosting) with DNS registration and a fictitious administrator (Jeremy Martin) were created to control the mirrors. A dedicated server was leased through The Planet (`www.theplanet.com`).

Setting up a public mirror for Debian, Ubuntu, CentOS, and Fedora involved acquiring the packages and metadata, creating an account on the distribution's website, and then notifying the distribution maintainers that the mirror is online. Ubuntu took over a month to check our mirror due to acknowledged but unspecified internal delays, but after checking our mirror approved it within a day. Debian and CentOS listed the mirror within a few hours; Fedora listed the mirror in minutes.

With Fedora, a few minutes after the mirror was officially listed, the mirror began receiving requests from Fedora clients. We believe this is the result of most Fedora

clients using `MirrorManager` [31] to dynamically select mirrors. `MirrorManager` has clients send requests to a central server that farms them out based upon the client IP, geographic location, country, etc. One interesting thing to note is that mirror administrators can specify an IP address range to which the mirror should serve packages. In fact, targeting a subnet means that clients in that subnet will use *only* that mirror. This allows easy targeting of attacks (to a specific country or organization) and reduces the number of other parties that will consume resources on the mirror.

3.4.2 Repository Key

While an attacker may be able to impersonate a repository, this may or may not mean that the attacker is able to create forged packages or metadata on the repository. Some repositories use a private key to sign the root repository metadata that is published by the repository. If the repository keeps its private key offline or is a mirror of another repository that does all of the signing, a compromise of the repository may not imply a compromise of the repository entity's key. A MITM that does not possess the repository entity's key also cannot sign package metadata.

In YUM and many APT distributions there is no repository key to compromise. Any attacker with control of a repository can launch these attacks without having to

compromise additional keys. If the repository signs package metadata, and has its key online then the attacker that has compromised the repository can forge package metadata. Similarly, an attacker that controls a third-party repository and lures an administrator into adding the repository key to her keyring can launch these attacks.

One item of interest is that since the root repository metadata and the signature of the root repository metadata file are in separate files, the root repository metadata and its signature are not downloaded and verified atomically. This means that in some cases the signature and root repository metadata file will not match (for example, as the files are updated) even in the absence of attackers. These false positives may help to mask attacks.

Note that even when the root repository metadata is signed and an attacker does not have the private key, an attacker can still launch package metadata attacks if the attacker can convince the repository to host a package the attacker created. This means that developers can trivially launch these attacks.

Although the root repository metadata is unsigned, one may argue that if the packages are signed, then the package metadata is protected from tampering. While YUM does not sign the root repository metadata, most YUM distributions do sign the packages themselves. One might assume that the package signature is used to verify the package metadata. This is not the case. The repository is trusted to

generate accurate package metadata that reflects the contents of the package. This package metadata is used by the dependency resolver to decide which packages should be downloaded. Thus, the dependency resolver is explicitly trusting the repository to provide accurate information. The only way to be sure that the repository provided accurate package metadata is to use the package metadata from the downloaded and verified package. This means that, in practice, the package signature is not used to verify the package metadata! As a result, package metadata retrieved from the repository about any package is (at best) only protected by the secure hash in the root repository metadata (signed using the repository key) and not the developer that packaged the software.

3.4.3 Package Key

In addition to keys used to sign the root repository metadata, some YUM-based distributions sign the packages themselves. If the key used to sign packages is compromised then an attacker can sign any package.

However, it is important to consider what happens when the dependency resolver decides a package that is unsigned or signed by an untrusted key must be installed to resolve dependencies. If an administrator willingly installs unsigned packages, then the result is equivalent to a private key compromise. With YUM, when an

administrator tries to install a package that is signed with a key that is not in her keyring, YUM will return an error if the `gpgkey` option is not set. If the `gpgkey` option is set, YUM will look for GPG keys at the URLs specified (often on the same repository) and ask the administrator if the administrator wants to install these keys in order to verify the package. So, if an attacker can provide its own key when the administrator requests the URL listed in the `gpgkey` option, all the attacker must do is provide the administrator a package with a signature the administrator cannot verify. The administrator will be prompted to add the attacker's key to her keyring!

In APT distributions and YUM distributions that do not sign packages, there is no package key to compromise. Repository and repository key compromises imply the ability to launch any attacks that would otherwise require a package key compromise.

3.4.4 Developer Key

In addition to considering the feasibility of attacks that circumvent the security mechanisms of package managers, it is important to consider how packages are legitimately updated on repositories. If an attacker can introduce malicious content onto a repository through legitimate channels then the attacker can also launch attacks.

The following describes the developer's package update process, using Debian as an example. To update packages, a developer uploads a package signed by her private key to a repository. If the signature is valid, corresponds to a key in the list of developers, and the package is correctly formatted, the package file is put into a pool of packages that form the daily update. All of the packages in the daily update have new package metadata generated for them. The root repository metadata file is signed automatically by a single Debian key that all Debian clients have in their keyring. The daily update is then pushed to hundreds of mirrors around the world that serve as package repositories.

There are several items of interest. First, the packaging and update process is fully automatic. From the description of security practices on the main Debian site [23], the only thing an attacker must do is compromise a developer's key to upload malicious packages.

Second, any developer can upload an updated version of any package. This means that you are implicitly trusting *every* Debian developer's key even if you don't use the software that developer maintains. This means that the distribution developer who is trusted to maintain `gnuchess` is also trusted for the `glibc` package. Considering that in November of 2007 there were more than 2448 keys listed in the Debian developer database, there are a significant number of keys that must be secure for the distribution to remain secure. Further compounding the problem,

there are keys that are as short as 768 bits and as old as 1993!

In addition to trusting developers that work on a distribution, it is important to consider third-party developers (developers that do not work on packages in the distribution). If an administrator wants to verify third-party software before installation, then the administrator must add the third-party developer's key to her keyring. Depending on the security measures undertaken by the third-party developers, this private key may be more or less secure than the administrator's distribution's private key. However, a third-party developer's key always represents another avenue of attack.

As a recent attack [75] has shown, it is feasible for package repositories to be poisoned with malicious versions of packages. An attacker obtained the password for a developer account. This account was used to upload a modified version of the SquirrelMail package with a flaw that allowed a remote attacker to execute arbitrary code. Interestingly, if the account compromised had access to the project website, the attacker would have been able to change the MD5 hash listed for the package and the attack may have gone unnoticed.

3.5 Attack Effectiveness

The previous section studied the feasibility of the attackers gaining the necessary access to launch these attacks on APT and YUM. This section studies the result of launching these attacks. A summary of the result of these attacks can be seen in Table 3.4.

3.5.1 Slow Retrieval

A simple attack is to allow the client to open a connection but then send data at a very slow rate. The attacker keeps the connection open for as long as possible. This prevents the client from installing updates from other repositories. APT and YUM don't log or print any useful information to help an administrator discover an attack has taken place. Other than "pausing" when the package manager tries to contact a repository, there is no output to indicate the problem.

3.5.2 Endless Data

Another attack is to return an endless stream of data to the client whenever files are requested (an attack described in other work [82]). This attack has an odd effect on

Attack Name	Description	Result
Slow Retrieval	An attacker slows repository communication so that package managers will "pause" and will not contact other repositories to retrieve package updates.	DoS
Endless Data	An attacker that controls a repository (or MITM attacker) returns an endless stream of data in response to any file request.	DoS / Crash
Replay Old Metadata	An attacker provides old metadata that is correctly signed (perhaps to prevent new packages from being considered)	Install An Outdated Package
Extraneous Dependency	An attacker changes the metadata for a package to indicate it depends on a package or packages of the attackers choice.	Install Any Signed Package
Depends on Everything	An attacker changes the metadata for a package to indicate it depends on everything.	DoS / Crash
Unsatisfiable Dependencies	An attacker causes a package manager to ignore valid packages because forged metadata indicates unsatisfiable dependencies	DoS / Outdated Package
Provides Everything	An attacker changes the metadata for a package to indicate it provides any dependency the client requests.	Install Any Signed Package
Use Revoked Keys	An attacker uses a revoked key to get clients to install packages	Install Any Package
Escalation of Privilege	An attacker compromises a key trusted for signing a specific group of packages and then gets clients to accept signed malicious versions of other packages.	Install Any Package

TABLE 3.4. This table lists attacks, a description of the attack and the result of a successful attack.

YUM and APT. Surprisingly, when YUM is given a `repomd.xml` file of unlimited size, YUM fills the disk and then exits silently with a 1 exit code — leaving the huge file on disk. Since no information is logged or printed about the error, this makes discovering the problem complicated (especially if YUM runs via auto-update).

APT is also vulnerable to this attack, but the size of the retrieved file is assumed to fit in a C unsigned long. This means that APT will willingly download up to 4 GB of data on 32-bit architectures for each file. If APT is compiled on a 64-bit architecture, then APT will happily try to download files greater than 18,000,000 TB! Interestingly, APT could prevent this type of attack in the case when the repository entity's key is not compromised since the root repository metadata and package metadata provide the data sizes of the files to be downloaded.

The endless data attack works to prevent clients from getting package updates from other repositories. However, this attack also consumes large amounts of disk space on the client as well as network bandwidth and CPU. Exhausting resources, especially disk space, on a client machine can have disastrous effects. For example, on Fedora and Ubuntu this prevents logging, corrupts databases, and halts mail delivery.

The damage caused by filling the file system can be mitigated by using a separate filesystem for the cache directory used by APT or YUM. However, this attack still

prevents updates from other package repositories from being installed.

3.5.3 Replay Old Metadata

Some APT repositories sign the root repository metadata to prevent tampering by an attacker. The signature prevents an attacker that cannot sign the root repository metadata from substituting arbitrary metadata. However, the signature does not prevent an attacker from replaying old metadata. An attacker may, for example, capture the root repository metadata from a date when a vulnerable package was released. At some time in the future, well after the problem has been fixed, the attacker may replay the captured metadata, causing clients that request the package to install the vulnerable version.

In APT, even though the metadata is signed to prevent tampering, there is no protection against replaying old metadata. APT ignores the date listed in the metadata file. The date in the root repository metadata could be older than the previous root repository metadata file, or a date in the future, and there is no complaint. In fact, APT overwrites the existing root repository metadata file and metadata with the files it is downloading. This means that unless the administrator retains the old root repository metadata file and metadata manually, the administrator has no way to check what the previous state of the repository was.

For YUM repositories and APT repositories that do not sign the root repository metadata, there is no need to replay metadata. Since no entity's signature protects the root repository metadata, an attacker can create metadata of its choosing.

3.5.4 Extraneous Dependencies

This attack is launched by an attacker that rewrites the dependency information of a package to say that the package also depends on another package. For example, an attacker can say that every package depends on some "extraneous" package of the attacker's choice.

This attack works on both APT and YUM. Neither APT nor YUM verify that the dependencies listed in a downloaded package matches the package metadata that was retrieved from the repository! The only restriction in the choice of extraneous package is that if package signatures are checked then the resulting package must be correctly signed. Even when signatures are used this attack can result in existing packages being downgraded to vulnerable versions or new, vulnerable packages being installed.

3.5.5 Depends on Everything

Another potential attack involves returning package metadata that states that a requested package has a huge number of package dependencies. In APT and YUM, all of these packages are downloaded before any signatures are checked.

In addition to consuming disk and network resources on the client, this attack can be used by an attacker that controls a repository to launch an attack on other repositories. To launch an attack on other repositories, the attacker's repository can advertise a new version of a package that depends on the entire set of distribution packages and that the attacker's repository hosts none of the distribution's packages. Assuming the client is configured to use multiple repositories, the client will download all the distribution's packages from other repositories.

3.5.6 Unsatisfiable Dependencies

To prevent installation of a package, an attacker can return a list of dependencies that indicate that the package has unresolvable dependencies. This prevents APT and YUM from installing the package, while to the administrator it appears that the repository or packager was in error, instead of an attack.

Note that the attacker need not be able to correctly sign the package. Packages are

not downloaded until after dependency resolution and signatures cannot be checked without the package. This means the client will not download the package and therefore will not check the signature.

3.5.7 Provides Everything

Another potential attack involves returning metadata that indicates a package resolves a huge number of virtual dependencies. Any time a package is needed to resolve a virtual dependency, this package will be considered. This package's metadata can be created by an attacker to cause the package to be installed over other packages that provide the same virtual dependency.

If there is a real package with the same name as a virtual dependency, the real package is always preferred. This allows an attacker to create a package `httpd` on its repository that will be preferred over any of the packages from the distribution that provide this virtual dependency (such as `apache`). When there are multiple packages that provide the same virtual dependency, APT resolves dependencies on virtual dependencies by choosing the package whose name comes first alphabetically. YUM chooses the package whose name has the shortest length. So an attacker can create a file named `a.deb` or `a.rpm` that will be preferred over all other packages.

This attack can be used by attackers in different ways. If package signatures are

used then an attacker can use this as another method to install extraneous packages. In some situations this is more effective than the extraneous dependencies attack because this will cause package installation for dependencies on other repositories. If package signatures are not used or the attacker has compromised the package signing key, then an attacker can have a malicious package of the attacker's choice be installed more frequently. Clients that want to install other software will have the malicious package installed as well. This attack can also be used to launch depends-on-everything attacks if the package that provides everything has a huge dependency list.

3.5.8 Use Revoked Keys

One common need in any program that uses public key cryptography is a mechanism for revoking keys. Package managers are no exception. There are two issues to consider: the number of keys and the method of revocation.

Unfortunately, the popular APT and YUM distributions that use signatures use a few keys to sign everything. In Debian and Ubuntu, all of the root repository metadata files are signed by the same key. In Fedora every package is signed by the same key. There is no way to retroactively revoke trust in a signed item, without revoking trust in all items signed by that key. Considering that there are many

cases in which security vulnerabilities in a released package are later discovered, revoking trust in a previously signed package is important.

The typical mechanism for revoking a key is by a notification in an ad hoc, out-of-band manner that a key should be revoked [17, 38, 42]. For example, such notification might take the form of a security announcement via email from an organization such as CERT [17, 38]. The administrator then manually removes the key from her keyring.

A package manager is designed to update packages rapidly and often automatically, which is at odds with the comparatively slow process of manual key revocation. Key revocation creates a race between the revoker (who is trying to remove trust in the compromised key) and the attacker (who is trying to use the key before it is revoked). Since key revocation is a slow manual process while updating packages is rapid, this gives the attacker a strong advantage.

3.5.9 Escalation of Privilege

YUM uses a set of trusted public keys to verify package signatures. To add a new public key for package verification, the administrator adds the key to her keyring. However, checking if a package's signing key is in an administrator's keyring is a true/false question — the key is either there or it is not. This means that an

administrator that wants to verify Apache project packages with the Apache project's public key will also implicitly (and silently) trust a `gcc` RPM signed by the Apache key.

YUM has a mechanism to mitigate these types of attacks. An administrator can specify that a repository should be used for a specific package or packages using the configuration option `includepkgs`. Similarly, a repository can be prevented from being used for specific packages with the `exclude` configuration option. However, this doesn't prevent an attacker that is trusted as a third party developer associated with one repository from infecting a client with malicious versions of a distribution's packages if the attacker can put its malicious packages on another repository the client uses. This is because the keys in the administrator's keyring that are used to verify packages are not tied to a repository or a set of packages. In order for an administrator to be able to use the `includepkgs` mechanism, the developer is required to set up a separate repository (a non-trivial commitment of time and resources).

An escalation of privilege attack exposes the tension between key management and package management. To understand the tension, consider an administrator that obtains and verifies Apache project's key but isn't planning on installing `apache` packages soon. If the administrator doesn't add the Apache project's key, then the administrator will need to verify the key is correct if the administrator decides to

install Apache later (an administrative pain). If the Apache project's key is added, then a compromise of the Apache key can impact the security of the client even if `apache` packages aren't installed.

3.6 Securing APT and YUM

There are several simple actions that mitigate the effectiveness of many of the attacks:

1. *Validate repository communication.* By checking that file sizes and data rates are reasonable when communicating with the repository, APT and YUM could limit the effectiveness of endless data and slow retrieval attacks.

2. *Track signature times.* APT (and YUM if it adds metadata signing) should refuse to accept older versions of signed data. This would limit the effectiveness of the replay old metadata attack by only allowing attackers to "freeze" the metadata at a certain date rather than replaying older metadata than the version the client has previously downloaded.

3. *Use HTTPS.* HTTPS makes it more difficult for an attacker to launch any of the attacks because a man in the middle will not be able to easily masquerade as a repository.

4. *Guard mirrors.* Delegating control of a mirror for a distribution should be treated with utmost caution. This will make it harder for an attacker to control a repository.

5. *Sign metadata and packages.* Signing both metadata and packages makes it

more difficult for an attacker to launch most types of attacks.

6. *Check that metadata is correct.* Once APT or YUM has decided to install a package, it should download the package, verify the package's signature, and verify that the metadata in the package matches the package metadata provided by the repository. This will help to prevent the depends on everything attack and extraneous dependencies attack when the attacker cannot correctly sign the package.

These measures will increase the difficulty in launching many types of attacks. However, within the architectures of APT and YUM there is no way to fix key revocation, escalation of privilege, provides everything, and unsatisfiable dependencies attacks. The security architectures of APT and YUM are fundamentally inadequate to address these issues.

3.7 Principles of Secure Package Management

Several areas must be addressed to provide package management security. This section discusses three principles for secure package management that provide this functionality. Then, examples of how these principles should be applied in practice are given.

3.7.1 Design Principles

Selective trust delegation allows an entity to trust another entity's signature only for specific packages. This allows an administrator to prevent escalation of privilege attacks by delegating the minimum amount of trust necessary. A hierarchical model is used in which an administrator may trust a project leader or distributor, that in turn trusts individual developers. This is fundamentally different from having the distributor sign a package simply because the developer signed the package. The distributor signaling trust in the developer is different because the distributor never signs the package, but instead signals trust in the individual developer's key. The distributor can revoke trust in the developer rather than requiring an administrator to revoke trust in the distributor's key. This provides a natural mechanism for key revocation and also removes the necessity of revoking a key to remove trust in a package.

Having customized repository views means that each client "sees" a different repository, which is actually an amalgamation of package metadata across all of the repositories the client is using. However, only package metadata signed by entities the client trusts is included. Since package metadata from untrusted entities is not included, package metadata from untrusted entities cannot impact the security of the package manager. This allows repository administrators to permit entities to freely add their own packages without compromising the security of the clients that do not trust those entities. Repository administrators then only need to worry about traditional problems for servers, such as preventing attackers from gaining root access and monitoring the disk space usage of individual entities, instead of being concerned with the validity of the contents of the packages or the package metadata.

Contrary to popular belief, there are subtle ways that package repositories are trusted by the package manager on a client. For example, package repositories are implicitly trusted to allow file downloads. Package repositories are also trusted to provide accurate and timely package metadata for the packages on the repository. Unfortunately, in many cases a man-in-the-middle can exploit these trust assumptions. By treating all interactions with the repository (and everyone in between) as though the interactions are with an untrusted entity, package managers can avoid these attacks.

3.7.2 Concepts and Examples

Selective delegation means that entities can choose to delegate trust to other entities in fine-grained ways. As a simple example, if Alice knows Bob maintains the `foo` package, Alice can trust `foo` packages that Bob trusts. This means that Alice will not trust a package `bar` that Bob trusts.

An example of how this might be used in practice is that the `foo` project entity has a key that is used to manage the project. Alice trusts the `foo` project entity to know which `foo` packages are valid. The `foo` project entity trusts the current developers on the project to know which `foo` packages are valid. A developer trusts a `foo` package using her personal, private key.

At first glance this may seem the same as having a single project key for signing, but there are several important differences. First of all, the project key is only used when project membership changes and so can be kept off-line (in contrast to signing every package with the project key). Second, no developer needs access to the project's private key. As developers come and go there is no need to change the project key used to sign releases. Third, revocation of a developer's key is as easy as the project key removing delegation to that developer. Administrators whose clients use the project's packages need not be aware that the project's membership has changed. Fourth, trust of individual packages can also be revoked without revoking

the signing key.

Not every administrator must make their own trust delegation decisions. Distribution maintainers would likely selectively delegate trust to projects, that would further delegate trust to developers. Distribution maintainers would request that the clients that use the distribution either trust the packages that the distribution entity trusts or delegate trust to the distribution entity.

Another example of the usefulness of selective delegation is indicating that a package is untrusted. There are several groups that monitor software vulnerabilities [17, 38]. Once a vulnerability is uncovered, these groups notify administrators that affected packages should not be trusted. Instead, these groups could ask administrators to delegate trust to them for knowing which packages to reject. An administrator can add this entity as an authority that knows which packages should not be trusted. As a result, the client will refuse to install a packages that is indicated as having a security vulnerability even if other entities indicate the package is trusted.

A related principle to selective delegation is a customized repository view. A customized repository view means that different clients that use the same repository may have a different view of the packages on that repository. For example, perhaps Alice trusts Bob to know about the `foo` packages, while Charlie does not. A client that Alice manages will have Bob's `foo` packages in its view of the packages in the

repository and a client Charlie manages will not.

Customized repository views extend to multiple repositories. A client may access packages and metadata on multiple repositories. This information is amalgamated to form a single, customized repository view with only data from the entities that the client trusts. If some of the repositories are down or have been compromised, the client will still be able to install packages from valid sources.

In providing customized repository views, it is important to keep track of the latest version of the files that have been retrieved. The client should never accept an older version of a file it has seen to protect against replay attacks.

To preserve security it is important to treat data coming from the repository as untrusted. There are many subtle issues that are important to consider. For example, dependency resolution is performed using package metadata. This information must be validated so that maliciously-modified metadata cannot change the behavior of package dependency resolution. Also, clients retrieve data from repositories to install package updates including security updates. If a client is using multiple repositories, it should not be possible for one of the repositories to prevent the client from installing packages from other repositories.

3.8 Security Architecture in Stork

Stork prevents the attacks by following the three design principles for package management security: selective trust delegation, customized repository views, and explicitly treating the repository as untrusted.

This section begins with a description of a mechanism to provide selective trust delegation called a *trusted packages (TP) file*. Then follows a description of signature wrappers (a signature with additional fields) that provide resilience against replay attacks to support customized repository views. Next, Stork's repository communication (which gathers the trusted packages files and package metadata) is described. Then the use of self-certifying path names to support both customized repository views and validate repository communication is described. A discussion of how dependency resolution with backtracking improves security is then provided. Tags are introduced to solve problems related to deciding which version of a package should be used to resolve a dependency. The section concludes with a discussion of how missing TP files are handled.

3.8.1 Trusted Packages

The primary architectural security difference between Stork and existing package managers is the addition of a new type of file called a trusted packages file (or TP file). TP files are used to provide selective trust delegation and also aid in supporting customized repository views. An example TP file can be seen in Figure 3.1.

An entity's TP file indicates which packages the entity considers valid. The TP file does not cause those packages to be installed, but instead indicates trust that the packages have valid contents and are candidates for installation. It is common for the distributor of a package to have multiple versions of the same package listed in her TP file so that clients can install older (perhaps more stable) versions of the package.

A trusted packages file allows an entity to delegate trust and specify individual package files that the entity trusts. In order to trust a package, the package name and the hash of the package's metadata are added to the trusted packages file. *This is not the same as adding the hash of the package to the trusted packages file as the metadata may now be verified independently of the package.* Since the package metadata contains a secure hash of the package, the package is still protected from tampering.

To delegate trust to an entity, the entity's name and public key are specified along with the packages and dependencies the entity is allowed to provide. This information is added to the trusted packages file. An entity can be trusted to know which packages to install (using `ALLOW`), which packages not to install (using `DENY`), or both (using `ANY`).

Lines in a trusted packages file are processed in order. The first line that a package matches classifies the package as either available for installation (allowed) or removes it from consideration (denied). Any packages that are unmatched are automatically rejected (there is an implicit `<FILE PATTERN="*" ACTION="DENY"/>` at the end). Thus, the order of lines in a TP file is important when determining which packages are candidates for installation.

Figure 3.1 shows a TP file without a signature wrapper. This TP file rejects any packages that the CERT entity rejects. This TP file specifically allows `emacs-2.2-5.i386.rpm`, several versions of `foobar`, and `customapp-1.0.tar.gz` to be installed if the package metadata matches the secure hash listed. It also trusts the Apache entity for packages named `apache*`. Additionally, the shown TP file indicates that the TP file of the Stork entity is used to determine trust of any package whose name starts with `stork` or `arizona`.

A common usage scenario is for an administrator to delegate trust to the

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes" ?>
<TRUSTEDPACKAGES>

<!-- Refuse any packages the CERT entity says are bad -->
<USER PATTERN="*" USERNAME="CERT" PUBLICKEY="MFwwD...4wEAAQ" ACTION="DENY"/>


<!-- Trust some packages that the administrator specifically allows -->
<FILE PATTERN="emacs-2.2-5.i386.rpm"   HASH="aed4...a732" ACTION="ALLOW"/>
<FILE PATTERN="foobar-1.01.i386.rpm" HASH="16b6...470" ACTION="ALLOW"/>
<FILE PATTERN="foobar-1.0.i386.rpm"  HASH="3945...7fd" ACTION="ALLOW"/>
<FILE PATTERN="simple-1.0.tar.gz"    HASH="2343...341" ACTION="ALLOW"/>

<!-- Trust the apache entity for apache packages -->
<USER PATTERN="apache*" PROVIDES="apache*,httpd" USERNAME="apache"
      PUBLICKEY="MFw_...EAAQ" ACTION="ANY"/>

<!-- Trust the 'stork' entity for stork and arizona packages -->
<USER PATTERN="stork*,arizona*" PROVIDES="stork*,arizona*" USERNAME="stork"
      PUBLICKEY="MFww...AAQ" ACTION="ANY"/>

</TRUSTEDPACKAGES>
```

FIGURE 3.1. **Example TP File.** This file specifies which packages and entities are trusted. Only packages allowed by a TP file may be installed. FILE actions are used to trust individual packages. USER actions delegate trust by specifying a entity whose TP file is included. Public keys and hashes are shortened for readability.

distributions and third party projects they use. The distribution's TP file selectively delegates trust to different projects for only the packages distributed by that project. The project's TP file delegates trust to the developers. The developer's TP files are used to trust individual packages.

3.8.2 Signature Wrappers

Stork uses signature wrappers to create the customized repository view that a client sees. This is the set of good metadata from all repositories the client references, including good information previously downloaded from the repositories. Package metadata is checked for validity (described in Section 3.8.4) and newer versions of files are selected. The resulting valid package metadata forms the customized repository view.

Customized repository views are formed using files retrieved from the repositories defined in the administrator's master configuration file. The newest valid versions of the `packages.pacman`, `groups.pacman`, and TP files are used. Additionally, only the package metadata trusted by the administrator's TP file (or TP files the administrator trusts) is considered. The resulting package metadata and newest versions of the other files comprise the client's customized repository view.

Signature wrappers protect TP files, `packages.pacman`, `groups.pacman`, master

configuration files, and the root repository metadata from tampering, prevent the replay of an older file than the package manager has previously used, and protect against an attacker permanently returning the same version of a file. Signature wrappers contain the timestamp, expiration time, signature, and a hash of the public key that was used to generate the signature. The timestamp prevents older versions of files from being considered over newer versions. The expiration time stops old files from being used indefinitely. The signature protects the file from tampering. The signature covers the expiration time and timestamp as well as the embedded contents. The public key is included for reasons explained in Section 3.8.4.

The timestamp specifies when the file was created. By default, the timestamp is generated from the time in seconds since the epoch. Clients track the latest version of each file the client has seen and will never accept an older version, thus preventing replay of old files.

Files can also be created with a negative timestamp. If a file has a negative timestamp, it means that the entity's key that signed the file should be treated as invalid. Any client with a file that has a negative timestamp will reject any file signed by the same entity's key regardless of timestamp. This is an effective way for an entity to indicate that its key has been compromised and that the key's signature should not be trusted. Negative timestamps were chosen to indicate a revoked key (as opposed to a separate field in the signature wrapper) to preserve backwards

compatibility with the existing signature wrapper format.

The signature wrapper also has an expiration time. This is checked against the time on the local client that uses the file. If the current number of seconds since the epoch is greater than the expiration time, then the file is invalid.

The expiration time mechanism requires that clients have roughly synchronized clocks. If this is a problem, an administrator can set a master configuration file option on a client to disable expiration checking but retain the signature verification and timestamp ordering. This allows administrators to permit clients with high clock skew to operate but reduces security (as there is no protection against an attacker continuously returning the same version of a signed file).

There are two comparisons that must be performed to determine which version of a signed file should be used. The expiration time of each file's signature wrapper must be examined to see if the file has expired and the timestamps in the file's signature wrappers must be compared to see which is newer. The order of the timestamp and expiration time comparisons may be relevant to which file is considered valid. For example, suppose that there are two files: an older file that has an expiration time in the future and a newer file that has already expired. If the expiration comparison is done first, then the older file will be used because the newer file has expired. If the expiration comparison is done after the timestamp comparison, neither file is

valid because the newest version has an expiration time in the past.

Stork performs the timestamp comparison first so that the file with the newest timestamp will always be used. This helps to prevent replay of old metadata because clients will never accept older files once the client has seen a newer file, no matter the situation.

3.8.3 Communicating with Repositories

Instead of a simply opening a repository connection for file download and waiting for download completion, Stork monitors the connection. If the connection is not transferring data above a configurable minimum rate, the repository is treated as dead and the connection is aborted. Stork also restricts the amount of data downloaded from a repository. If a repository responds with more than the expected amount of data or more than a configurable maximum then Stork drops the connection.

Stork supports multiple data transfer protocols, including HTTPS. Using HTTPS prevents man-in-the-middle attackers from masquerading as a repository.

3.8.4 Self-certifying path names

Along with signatures, Stork uses self-certifying path names [48] to detect tampering. These identifiers are present in the URLs or file names of TP files, `packages.pacman`, `groups.pacman`, master configuration files, packages, and package metadata. Self-certifying path names allow any party, such as a repository or nest, to check the validity of a file.

A package or package metadata is stored on a repository at a URL containing its secure hash. All packages and package metadata are considered to be immutable. Any entity can verify that a file has not been tampered with by checking the secure hash of the retrieved data. This allows a repository to check that uploaded packages and package metadata are valid. This is also used by the client to validate that information retrieved from a repository is correct.

A TP file has a public key embedded in its name instead when the key will fit given the filesystem restrictions on file name length. Using an embedded public key, a repository can verify the file is unmodified by checking that the TP file is correctly signed and that the public key in the secure identifier corresponds to the private key that signed the file.

On some file systems, keys with many bits are too long to be embedded in the file

names. In this case the public key is placed in the signature wrapper and a secure hash of the key is placed in the URL.

Since TP files are mutable, multiple valid copies of a TP file may be uploaded to a Stork repository. In this case, the Stork repository keeps the version with the latest timestamp. If any of the files specify that the key should be revoked (via a negative timestamp), then the file with the negative timestamp is retained.

Having a Stork repository keep the newest version of a file is an optimization that prevents attackers from replaying old copies of files. However, a repository that does not perform this action still provides security assurances to the clients it serves. Even if a repository colludes with an attacker to accept old files, Stork's use of expiration dates and timestamps limits the effectiveness of replaying old TP files. This means that a malicious repository can only "pause" an existing TP file while the TP file is within its expiration period and cannot cause a client to accept an earlier version. However, a well-behaved repository improves the security of the clients that use the repository by ensuring that new versions of files that are uploaded are available to clients.

Other metadata files (`packages.pacman`, `groups.pacman`, and master configuration files) are treated the same way as TP files. These files also have self-certifying path names.

3.8.5 Dependency Resolution

Another aspect of package management that has security implications is dependency resolution. An attacker may be able to block access to some repositories but not others, which means that actions by an attacker (or repository failures) may make some packages unavailable. Inability to access a package repository should not prevent package installation when alternate trusted packages that fulfill the same dependencies are available. If this is not the case then the failure of a few repositories can deny packages to many clients. In addition, when an administrator uploads a package to a repository it is possible that the administrator will forget to add a dependent package. If the administrator does not specify which version of a package should be added on a client, the administrator's client should not be prevented from installing another trusted version of the package.

For example, suppose there is a client that uses two repositories. The first repository hosts the version of `bar` that is preferred by the administrator's TP file (preference is discussed in Section 3.8.6). The second repository also hosts a version of `bar` that is also trusted. Suppose that the first repository is offline when the client tries to install `bar`. The dependency resolution mechanism should install the version of `bar` on the second repository. If the dependency resolution mechanism does not install `bar` then an attacker that can block access to the first repository

can stop the client from installing `bar`. If the administrator specifies a version of `bar`, that version will be the only one installed. If the administrator wants the version of `bar` on the second repository to be installed only until the first repository comes back online, the administrator can specify that `bar` should be updated in the `packages.pacman` file.

In order to perform dependency resolution and handle unresolved dependencies, the dependency resolution engine must support backtracking. Before explaining backtracking in dependency resolution, classical dependency resolution (the process used by APT, YUM, and other package managers) is first described. Classical dependency resolution is a process that is given a package (or packages) and is asked to provide a set of packages that when installed will have no unresolved dependencies. The steps that a classical dependency performs are:

1. The outstanding dependency list is initialized to contain the packages that the administrator has selected to have resolved.

2. The suggested packages list is initialized to be empty. The suggested packages list will contain all of the packages that must be installed to resolve the dependencies.

3. The first item in the outstanding dependency list is removed and set to be the "current dependency".

4. A package that satisfies the current dependency is chosen. If more than one package qualifies the choice is based upon the tie-breaking criteria of the package manager (typically something like shortest name or alphabetical). If there is no package that meets the criteria then the package manager prints an error and exits. The package that is chosen to meet the current dependency is the "current package".

5. The current package is added to the suggested packages list.

6. The dependencies of the current package are examined. Any dependency that is not installed and is not resolved by a package in the suggested packages list is added to the outstanding dependency list.

7. If the outstanding dependency list is non-empty then go to Step 3.

8. Return the suggested packages list.

The dependency resolution engine provided will resolve dependencies if each dependency can be resolved by the preferred package. However, if the preferred package cannot resolve a dependency, the dependency resolution process will fail. By adding backtracking, the dependency resolution engine will resolve dependencies if there exists a version of a package that can resolve each dependency and that package can have its dependencies resolved.

In order to add backtracking, the dependency resolution algorithm must be modified slightly in Step 4 to save the current state whenever there are multiple choices of packages to resolve a dependency. Step 4 also must be changed to backtrack to try other packages whenever the package chosen results in an unresolvable dependency. When backtracking the suggested packages list and outstanding package lists must be restored to the state they were in when the choice was made.

3.8.6 Tags

Another issue with dependency resolution is which package should be preferred to resolve a dependency when multiple packages may do so. Originally in Stork, packages were preferred based upon the order of the line that allowed the package in the TP file (the order is still used as a tie-breaker for other preference mechanisms). Preference based upon order in the trusted packages file works well for an administrator that wants to add preferred versions of packages early in the trusted packages file to override the package decisions made by other entities the administrator trusts. However, there are two situations in which the order in the TP file is problematic. First, if an administrator manages clients that use different distributions, the administrator would like each client to prefer (or possibly to only use) the packages from the distribution the client uses. Second, a project may have many developers. The project entity would like to signal trust in all of the

developers and have the preferred version of the project's package be the latest version produced by any developer. If the preferred version is order based, then the preferred version of the package will be the first version in the TP file of the first developer trusted by the project entity.

To resolve the problems with order based dependency resolution preferences, Stork provides *tags*. A tag is added to an entry in a TP file to signal that the package has a specific property. For example, an entity can add a tag that signifies that a package belongs to a specific distribution. An administrator can then specify that a client should only install packages that have that tag (and are thus from that distribution). Administrators can also list preferences for tags so that the packages in one distribution will be preferred over another distribution, but both distributions are acceptable. An administrator may want to use a single TP file for clients that have different distributions. There administrator can specify that the preferred tag is `%ARCH%`. The `%ARCH%` tag is replaced by the name of the client's distribution. The `%ARCH%` tag will ensure that the distribution's packages are used instead other versions of packages.

To support project entities in which preference should be decided based on the order in which packages are added, *timestamp* tags were added. An entity that adds a package can add a timestamp tag to a package. The package with the latest timestamp tag will be preferred over other packages that fulfill the same

dependency. Timestamp tags allow a project key to delegate trust to different developers and have the developer that puts the latest timestamp on a package have that package version be preferred.

If tags are not used or more than one package has the same tag, then the preference is determined by the order the packages are listed in the trusted packages file.

3.8.7 Missing TP Files

There are four logical things that the Stork package manager might do when a TP file is missing. Unfortunately, each is wrong in some scenarios and right in others. Given the information that the package manager has, it is not possible to make the correct decision all of the time. As a result the package manager provides a choice of behaviors to the administrator.

The four actions that the Stork package manager can take are:

1. *Ignore.* Ignore the fact that the TP file is missing. Treat the line that delegates trust to the entity as though the line was not there.

2. *Deny.* Treat a missing TP file as though the missing TP file contains the action <`FILE PATTERN="*" ACTION="DENY"/`>. This will act the same as ignore for entities that are delegated trust using `ALLOW`. When the package

action was `ANY` or `DENY` then a package that matches the pattern will be denied.

3. *Denyall.* Treat the line that includes the TP file as though the line is a `DENY` for any package that matches the pattern. This will have the same effect as Deny when the action was `ANY` or `DENY`. However, this will also prevent the installation of packages that match an `ALLOW` line.

4. *Exit.* The package manager prints an error message and refuses to perform any package management actions.

To illustrate that each of these is the wrong (and right) thing to do in some scenarios, an example TP file is shown in Figure 3.2. Assume that there is no TP file for the `foo` project or `security-team` entity on the client. The administrator requests the installation of `foo`, `bar`, and `baz`. Suppose `foo` depends on `bar` and `bar` depends on `baz`. Four examples are given to show how different answers can be correct depending on the situation.

1. The TP files are missing because an attacker is trying prevent the client from installing the packages (i.e. a denial-of-service attack). In this case, the administrator wants a valid version of `foo`, `bar`, and `baz` installed, regardless of the missing TP files. Since the administrator lists versions of `foo`, `bar`, and `baz` in the TP file, these versions should be installed.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes" ?>
<TRUSTEDPACKAGES>

<!-- Trust the 'security-team' entity to know about bad foo packages -->
<USER PATTERN="foo*" USERNAME="security-team" PUBLICKEY="MFskx...j34XEjAQ" ACTION="DENY"/>

<!-- Trust a package that the administrator specifically allows -->
<FILE PATTERN="foo-1.0-1.rpm" HASH="28934...3974" ACTION="ALLOW"/>

<!-- Trust the 'bar' entity for bar packages -->
<USER PATTERN="bar*" PROVIDES="bar*" USERNAME="bar" PUBLICKEY="MFskx...4XEjAQ" ACTION="ALLOW"/>

<!-- Trust a package that the administrator specifically allows -->
<FILE PATTERN="bar-1.0-1.rpm" HASH="835e6...d341" ACTION="ALLOW"/>

<!-- Trust a package that the administrator specifically allows -->
<FILE PATTERN="baz-1.0-1.rpm" HASH="b2346...ffde" ACTION="ALLOW"/>

</TRUSTEDPACKAGES>
```

FIGURE 3.2. **Example TP File for Missing Packages.** This TP file is used to provide examples of how the package manager cannot know the correct action to take when a TP file is missing. Public keys and hashes are shortened for readability.

2. The TP files are missing because an attacker is trying to prevent the client from getting the `security-team` TP file. The `security-team` TP file may deny the version of `foo` that the administrator has specified in the TP file due to a security problem with `foo`. In this case, the administrator wants a valid version of `bar` and `baz` installed but `foo` should not be installed.

3. The TP files are missing because an attacker is trying to get the client to install the outdated version of `bar` in the administrator's TP file. The attacker may want the client to install the outdated version of `bar` because of security problems with it that are resolved in the version of `bar` listed in the `bar` project's TP file. In this case, the administrator wants a valid version of `baz` installed but `bar` and `foo` should not be installed.

4. The TP files are missing because an attacker is trying to get the client to install `baz` without current versions of `foo` and / or `bar`. This may be the case because the newest versions of `foo` and / or `bar` provide logging or attack detection functionality that the attacker may trigger while attacking `baz`. In this case the administrator doesn't want any packages installed. The administrator also is likely to want to be warned of the missing TP files without any packages being installed if the administrator is running `stork` by hand.

The examples given correspond to the different options as is shown in Table 3.5.

	Package Manager Action			
	Ignore	Deny	Denyall	Exit
Example 1	Correct	Omitted `foo`	Omitted `foo` and `bar`	Omitted `foo`, `bar`, and `baz`
Example 2	Installed `foo`	Correct	Omitted `bar`	Omitted `bar` and `baz`
Example 3	Installed `foo` and `bar`	Installed `bar`	Correct	Omitted `baz`
Example 4	Installed `foo`, `bar`, and `baz`	Installed `bar` and `baz`	Installed `baz`	Correct

TABLE 3.5. The correspondence between the examples and choice of how the package manager handles missing TP files is shown. Each example is correctly handled by one of the ways to deal with missing TP files and wrong in the others. The packages omitted or additional packages installed are listed.

Each configuration option works perfectly for one example, but does the wrong thing for the other examples. Since the Stork client tools cannot know what the right answer is for the situation, there is a configuration option that controls the behavior. An administrator can choose the option that specifies the behavior she wants her client to have.

3.9 Attack Effectiveness in Stork

By following the principles for secure package management, Stork has worked to mitigate the attacks that are effective on APT and YUM. This section revisits these attacks to examine their effectiveness on Stork.

3.9.1 Slow Retrieval

Communication monitoring by the client prevents slow communications from "pausing" the package manager. If a connection is too slow, the transfer is aborted and the repository is treated as down.

3.9.2 Endless Data

Stork protects against endless data attacks. In some cases a Stork client does not know the correct size of data it is downloading: when an attacker controls a repository (and can arbitrarily set the size), and when retrieving the initial root repository metadata. There are maximum file sizes for every type of file downloaded as well as a maximum size for a repository as a whole.

In the case of an uncompromised repository, all files other than the root repository

metadata are of a known size. Stork verifies that the files it downloads do not exceed that size.

3.9.3 Replay Old Metadata

Since the root repository metadata, TP files, `package.pacman`, `groups.pacman`, and master configuration files have a signature wrapper, an attacker cannot replay older files than the client has already seen. Also, the expiration time prevents files from being used indefinitely.

3.9.4 Extraneous Dependencies

This attack cannot be launched by an attacker that compromises a repository since the developer's signature protects the package metadata. The key difference between the Stork client and existing package managers is that, with Stork, the *developer's key* protects the package metadata, not the *repository's key*.

3.9.5 Depends on Everything

Similar to extraneous dependencies, this attack cannot be launched by an attacker that compromises a repository since the developer's signature protects the package metadata.

3.9.6 Unsatisfiable Dependencies

This attack is prevented for the same reason as extraneous dependencies: a repository compromise does not imply the ability to forge package metadata. However, if a package legitimately has unsatisfiable dependencies, the dependency resolution mechanism will find a trusted version of the package and install that version instead.

3.9.7 Provides Everything

This attack cannot be launched by an attacker that compromises a repository since the developer's signature protects the package metadata. The other consideration is whether or not an attacker can cause a package the attacker is trusted to provide be installed when not requested. In Stork an entity can prevent this attack by

restricting the package metadata signed by a specific entity to only provide certain dependencies.

3.9.8 Use Revoked Keys

Delegation using a trusted packages file provides a mechanism for revoking trust in a package or revoking trust delegated to an entity. Assuming that project keys are used only to change the list of developers trusted to sign project files (as discussed in Section 3.7.2), the most likely avenue for compromise is an individual developer key. These can be revoked individually by the project key without involving other entities. If a project key is compromised, a trusted packages file with a negative timestamp can be created so that clients will not continue to use the compromised key.

Unlike APT and YUM, the speed of revocation in Stork is the same as the speed of trust. This means that individuals that revoke trust are not at a disadvantage because of the inherent speed of the mechanism used to do so.

3.9.9 Escalation of Privilege

Trusted packages files allow very fine-grained trust decisions to be delegated to other entities. This means that if an administrator trusts Apache to know about the validity of `apache` packages but never installs an `apache` package, an Apache key compromise will not compromise the security of the administrator's clients. This removes the administrator's dilemma about whether to add keys the administrator can verify when the package may not be needed (from Section 3.5.9). The simple and secure answer for Stork administrators is to trust all of the project keys that the administrator knows are valid but to trust them only for packages provided by that project. By using selective trust delegation, only clients that install a given project's packages are at risk if that project's key is compromised.

3.10 Related Work

There are many package managers for Linux including Slaktool [72], pacman [7], YaST [90], Urpmi [81], and emerge [63]. These package managers have flaws similar to those in APT and YUM. Slaktool and pacman do not attempt to provide security and are widely known to be insecure. Urpmi and emerge have security models very similar to YUM. YaST has a security model similar to APT when using package signatures. The attacks in Section 3.2 effect Urpmi, emerge, and YaST in similar ways to APT and YUM.

Popular BSD package managers are `pkg_add` [13] and `freebsd-update` [60]. `pkg_add` relies on administrator to check that detached signatures match the provided tarball, however these signatures are unavailable in practice and so are not widely used. `freebsd-update` signs the root repository metadata and does not sign packages or use HTTPS making it similar to APT from a security standpoint.

Many dependency resolvers have nice front-ends, including GUIs in many cases [44, 78]. These GUI-based tools are usually just a different interface to the functionality provided by a command line dependency resolver and so are not relevant from a security standpoint.

Another way to update software on an operating system is a software update

system. There is a recent study [9] of the security of software update systems. Different software update systems have different properties, but many popular software update systems have security problems in how they authenticate data and communications. Regardless software update systems do not support dependency resolution or selective delegation and as such are unacceptable for many situations.

There are a variety of ways to update and install software on modern operating systems besides package managers. There are systems that ensure the authenticity and integrity of software (including SFS-RO [48], SUNDR [49], Deployme [53], Self-Signed Executables [89]), software installers [41, 88], and code signing certificates [21]. These systems do not support selective delegation or protect dependency information. These systems are meant for cases in which every administrator knows the organization that is supposed to be distributing each piece software and there are no software dependencies — unrealistic requirements for many scenarios.

3.11 Conclusion

This chapter explored package management security by first examining nine attacks that existing package managers are vulnerable to and the common themes behind those attacks. Those attacks were shown to be feasible and allow attackers to install malicious packages, prevent legitimate package updates, and crash the client. While some attacks can be mitigated with simple patches to existing package managers, many flaws are architectural in nature. Three design principles for constructing an architecture that does not have these flaws were presented and implemented in Stork. Stork is not vulnerable to the attacks that affect other package managers.

Chapter 4

Sharing

The popularity of virtual machine environments such as Xen, VMWare, and VServers has created new challenges in package management. These systems attempt to isolate different virtual machines so that they function as independent systems. Unfortunately this isolation leads to inefficiencies in package management.

There are two major efficiency problems that VM environments create. First, isolation leads multiple VMs that want the same package and reside on the same physical machine to download the package separately. This wastes network bandwidth. Second, multiple VMs on the same physical machine install the package separately. This means there will be separate copies on disk which may also lead to increased memory use.

This chapter describes how Stork uses the nest to break the isolation provided by VM environments to improve efficiency. The nest improves efficiency by sharing files between VMs. The nest downloads packages only once per physical machine and securely shares installed packages between VMs. Sharing is evaluated by

constructing two different architectures for sharing. These architectures demonstrate that the nest presents trade offs of security versus ease of implementation.

4.1 Map

This chapter describes how Stork improves the efficiency of package management in VM environments by sharing files between VMs on the same physical machine. First, some examples describe inefficiencies in VM environments that can be resolved using sharing (Section 4.2). Then the potential benefits of sharing are analyzed to determine if sharing is worthwhile (Section 4.3). Once the benefits of sharing are established, Section 4.4 provides an overview of how Stork uses sharing to improve efficiency. Section 4.5 discusses the underlying mechanisms for sharing on different architectures. Section 4.6 revisits the client tools (previously described in Section 2.5) to examine the VM's actions to facilitate sharing. Section 4.7 describes the nest in detail, including the share (Section 4.7.1) and prepare (Section 4.7.3) functionality managers. Section 4.8 walks back through the examples provided in Section 4.2 to describe in detail how sharing works in practice. Section 4.9 discusses different locations where the nest can reside and the implications this has on security and ease of implementation. Section 4.10 examines how the sharing provided by the nest in Stork differs from other techniques for sharing. Section 4.11 concludes.

4.2 Examples

This section provides examples of inefficient actions performed by package managers in VM environments. The underlying theme that these actions have in common is that unnecessary duplication leads to inefficiency.

Later, in Section 4.8, these examples are revisited using Stork. Stork uses sharing to remove unnecessary duplication, which makes the examples in this section more efficient.

4.2.1 Duplicate Downloads

Suppose multiple VMs on the same physical machine are served by the same package repository. When a VM wants to download metadata or a package, the VM contacts the repository and downloads the file. This file may have already been downloaded by a VM on the same physical machine. These duplicate downloads unnecessarily consume network bandwidth.

However, this problem is not unique to files downloaded from a single repository. Two VMs on the same physical machine may also use different package repositories but download the same file from those repositories. Ideally, the duplicate downloads would be prevented even if the file is downloaded from different repositories.

4.2.2 Duplicate Package Files

Multiple VMs on the same physical machine have installed the same package. Each VMs has its own copy of the package on disk, leading to increased disk usage.

In addition, many operating systems and VMMs are able to reduce memory usage when multiple instances of the same executable are running at the same time. If each VM's executable is in a separate file, the VMM may not detect that the executables are the same. Since the executables are considered separate, the VMM is not able to reduce memory usage.

4.3 Benefits of Sharing

As the examples have shown, there are two main areas in package management in which sharing is beneficial. The first is removing duplicate downloads. Duplicate downloads occur when multiple VMs download the same package or metadata. Ideally, duplicate downloads would not occur and VMs on the same physical machine share the same file that would have been retrieved by separate downloads.

The second area in package management in which sharing is beneficial is sharing the files in packages between VMs. This means that there will only be a single copy of a package's files on disk no matter how how many VMs install the package. This increases the efficiency of disk and memory usage.

4.3.1 The Benefit of Removing Duplicate Downloads

This section examines the benefit of preventing duplicate downloads. To better understand the benefit, it is important to know how often different VMs on the same physical machine will download the same data. If VMs on the same physical machine never download the same data, then there is no potential benefit for duplication removal because there are no duplicate downloads. If nearly all of the files downloaded by VMs on the same physical machine is duplicated then the

impact of removing duplicate downloads is high.

To examine the frequency of common package selection, a snapshot of the `packages.pacman` files on the main Stork repository was examined on February 12th, 2008. In the 91 `package.pacman` files on the main Stork repository, administrators had 465 package actions. Of the 125 unique packages that had package actions, 60 of those packages were selected by a single administrator. However, the five most popular packages had actions from at least 10 administrators. On average, each package had 3.72 actions. These results demonstrate that there is a significant overlap in package selection among administrators. Since many administrators select the same packages there are many duplicate package downloads. This means the removal of duplicate package downloads will reduce the packages downloaded by 73%.

In addition to packages, package managers download metadata from a repository. It is common for either a distribution or organization to use a single repository (as Stork and PlanetLab do) or to have mirrors that are exact copies of a central repository. Either way the downloaded metadata is the same for all VMs that use the repository. If all VMs run the same operating system (as is the case in VServers and PlanetLab), this makes it likely that the metadata downloaded by every VM will be the same.

Thus, all VMs on a physical machine are likely to use the same repository or mirrors and therefore download the same metadata. This means that if VMs share metadata, there will be one metadata download per physical machine and all VMs will share the same metadata. According to CoMon [59], on average there are more than 50 VMs per PlanetLab physical machine so downloading metadata once per physical machine should have about 1/50 of the cost of downloading metadata once per VM.

4.3.2 The Benefit of Sharing Package Files

This section examines the benefit of sharing packages files. First, this section examines disk space savings when sharing packages in VM environments. Then, this section measures the memory savings provided by processes in shared packages that are run at the same time in multiple VMs.

4.3.2.1 Disk Usage The first experiment measured the amount of disk space saved on PlanetLab physical machines by installing packages in a shared manner versus installing packages in VMs individually (Table 4.1). These measurements were collected using the 10 most popular packages on a sample of 11 PlanetLab physical machines. Some applications consist of two packages: one containing the application

Rank	Package Name	Disk Space (KB)		Percent Savings
		Standard	Shared	
1	scriptroute	8644	600	93%
2	undns	13240	652	95%
3	chord	64972	1216	98%
4	j2re	61876	34280	45%
5	stork	320	32	90%
6	bind	6884	200	97%
7	file	1288	36	97%
8	make	808	32	96%
9	cpp	3220	44	99%
10	binutils	6892	60	99%

TABLE 4.1. **Disk Used by Popular Packages.** This table shows the disk space required to install the 10 most popular packages installed by VMs on a sampling of PlanetLab physical machines. The *Standard* column shows how much per-VM space the package consumes if nothing is shared. The *Shared* column shows how much per-VM space the package requires when installed files are shared.

and one containing a library used exclusively by the application. For the purpose of this experiment the packages are treated as a single package.

To test the benefit of sharing, each package was installed in multiple VMs. The packages examined here are RPMs which have information describing which files are documentation, configuration files, or application data. Application data and documentation files were shared between VMs after checking to ensure that the files are the same on each VM. Configuration files are not shared because they are likely to be modified by a VM and shared files are unable to be modified. More information about why configuration files are not shared is provided in the discussion about the sharing mechanism in Section 4.5.

For all but one package, sharing reduced the per-VM disk space required to install a package by over 90%. The actual disk savings depend on the number of VMs that install the package. The first VM that installs a package has no space savings and takes the standard amount of disk space. Every VM that installs the package thereafter incurs only the shared size instead of the standard size.

One package, `j2re`, had savings of only 45%. This was because many of the files within the package are themselves inside of archives. The post-install scripts extract these files from the archives, and are run separately by each VM so the extracted files are not shared between VMs. By repackaging `j2re` so that the extracted files

are part of the package, this issue can be avoided.

4.3.2.2 Memory Usage When two processes that use the same executable run at the same time, the memory used is not double that of a single process using the executable. This is because only one copy of the executable's code and libraries are stored in memory. These memory savings also occur when multiple VMs run processes that use an executable that is shared between VMs.

Table 4.2 shows the amount of memory shared when different VMs run the same executable. To examine the amount of memory saved, the primary application from each package in Table 4.1 was run and its memory usage was analyzed. The `pmap` command was used to dump the process's address space. Using the page map data, it is possible to classify memory regions as shared or private. This results in an accurate assessment of how much of the virtual address space of the processes is shared. However, the results do not necessarily correspond exactly to the amount of memory shared as some pages in the virtual address space may not be resident in memory (perhaps due to paging).

Another difficulty in measuring memory use is that the amount of memory used may change as the process runs. Daemons were simply started and measured. Applications that process input files (such as `java` and `make`) were started with a minimal file that goes into an infinite loop. The remaining applications were asked

Rank	Package Name	Application Name	Memory (MB)		Percent Savings
			Standard	Shared	
1	scriptroute	srinterpreter	5.8	3.2	45%
2	undns	undns_decode	4.2	2.0	53%
3	chord	adbd	7.6	2.3	70%
3	chord	lsd	7.5	1.1	86%
4	j2re	java	206.8	169.5	18%
5	stork	stork	3.4	1.2	64%
6	bind	named	36.7	32.1	12%
7	file	file	2.6	1.3	50%
8	make	make	2.5	1.1	54%
9	cpp	cpp	2.5	1.2	52%
10	binutils	objdump	3.3	1.4	59%
10	binutils	strip	2.9	1.0	65%
10	binutils	strings	3.4	1.7	50%

TABLE 4.2. **Memory Used by Popular Packages.** Packages that are shared allow VMs to share process memory. The *Standard* column shows how much memory is consumed by each process when nothing is shared. With shared processes, the first process will consume the same amount as the *Standard* column, but additional processes only require the amount shown in the *Shared* column.

to print their usage information and the memory used was measured before the application exited.

The resulting measurements show that sharing typically reduces the memory required by additional processes that use the same executable by 50% to 60%. There are two notable exceptions: `named` and `java`. These processes allocate huge data areas that are much larger than their text segments and libraries. Data segments are private, so this outweighs any benefits in sharing text and libraries.

4.3.3 Summary

This section has shown there is considerable room for improving efficiency in package management by sharing between VMs on the same physical machine. On average a package is downloaded 3.72 times, demonstrating that preventing duplicate package downloads will reduce package downloads by about 73%. Also, metadata downloads can be reduced to one per physical machine instead of one per VM which reduces the cost by a factor of 50. Disk space savings by sharing will typically be about an order of magnitude (minus packages that unpack or build executables during the post install scripts). The memory saved by sharing executables is about 50%. These results show that sharing is worthwhile in VM environments.

4.4 Sharing Overview

As Chapter 2 discussed, Stork shares files using the nest. A nest process runs on each physical machine that supports VMs. When the client tools run in a VM, the client tools communicate with the nest on the physical machine to coordinate sharing.

Before sharing occurs, the VM and the nest authenticate each other regardless of the architecture. The VM can authenticate the nest because only the nest is allowed to bind to a socket on a specified low numbered port due to protections provided by the VMM. The nest authenticates the VM using sharing primitives which also leverage protections provided by the VMM.

In order to prevent duplicate downloads, the client tools have a transfer specialized plug-in that performs a nest "transfer" of files. However, the specialized plug-in does not communicate with the repository, but instead asks the nest for the file. The nest keeps a cache of previously downloaded files. If the nest has the file in its cache, the nest shares the file with the VM. If the nest does not have the file in its cache, the nest downloads the file, adds the file to the cache, and then shares the file with the VM. The cache on the nest thus prevents duplicate downloads of the same file.

The nest also shares files in packages between VMs. To do this, the package

functionality manager used by `stork` has a package specialized plug-in that installs shared RPM packages. Instead of the files in the package installing normally, the nest overwrites the VM's copy of the files with hard links to shared versions of the files. The shared files are protected so that the VM cannot modify the file. This allows multiple VMs to share the same package files without compromising security.

4.5 Sharing Mechanism

This section describes the mechanisms used to securely share files on VServers (Section 4.5.1) and PlanetLab (Section 4.5.2) architectures. The different operations necessary to share files are also discussed.

Sharing is performed between VMs on the same physical machine in different ways on the PlanetLab and VServers architectures. However, the underlying functionality that the VMM provides to the nest is similar. This functionality consists of four different actions: binding a socket of the nest to a well known port so that a VM can authenticate the nest, the ability to protect a file so that a VM cannot modify it, the ability to share a directory between VMs, and the ability to share a file between VMs.

4.5.1 VServers

On the VServers architecture, the nest runs in the VMM. The VMM has access to the file system of each VM because the file system for a VM is a subdirectory in the VMM's file system.

The nest binds to a well known, low numbered port (< 1024) so that a VM can authenticate the nest. A VM in the VServers architecture does not have the

permissions necessary to bind sockets to port numbers less than 1024. The nest is in the VMM and so may bind a socket to a port less than 1024. Thus a VM can authenticate the nest by connecting to the port since only the nest can bind a socket to the port.

A file is protected by setting its IMMUTABLE_LINKAGE_INVERT_BIT. This prevents any VM from modifying the file (although a VM may unlink the file). A VM does not have the appropriate permission to remove the IMMUTABLE_LINKAGE_INVERT_BIT. This bit is set on any file shared between VMs to prevent an attacker from changing the file. Protected files may be removed but not modified in place. A protected file may be modified by copying the file, editing the copy, and then replacing the existing version of the file in the VM. This results in the VM having a separate, unshared version of the file that is not protected.

Sharing a directory between VMs is done using the `mount` command. The `mount` command takes an option `--bind` that allows a directory in a file system to be mounted as though it is a separate file system. This means that the nest can use the `mount` command to share a directory between VMs. The nest is in the VMM and is assumed to be trusted, so the nest is allowed to share whatever directories it requests.

Sharing a file between VMs is straightforward on the VServers architecture. The VMM (where the nest runs) has access to the file system of the VM. The nest can share a file with a VM by hard linking the file into an appropriate location in the VM's file system.

4.5.2 PlanetLab

On the PlanetLab architecture, the nest runs in a VM. The nest uses a service Proper [51] to perform privileged operations. Proper checks that an operation the nest requests is allowed and only permits authorized actions.

As in the VServers architecture, the nest binds to a well known, low numbered port (< 1024) so that a VM can authenticate the nest. A VM in the PlanetLab architecture does not have the permissions necessary to bind a socket to a port numbered less than 1024. The nest runs in a VM which means the nest cannot bind a socket to a port numbered less than 1024. However, the nest can bind a socket to a low numbered port through Proper. Proper checks that the slice of the nest VM has permission to bind a socket to the specified port. The port that the nest uses is well known so all VMs have this information. A VM knows that it is talking to the nest by connecting to the port since Proper prevents other VMs from binding to that port.

Protection of a file is accomplished by requesting that Proper set the NOCHANGE bit for the file. This bit functions in an identical way to the IMMUTABLE_LINKAGE_INVERT_BIT described in the VServers architecture section. Before Proper grants the request, Proper checks to see if the nest VM's slice has permission to set the NOCHANGE bit. The nest slice has been granted this permission by PlanetLab's administrators. Thus Proper will set the NOCHANGE bit on the files in the nest VM when the nest requests the action. As with VServers, an administrator that wants to change a protected file can copy the file, edit the copy, and then replace the existing version.

Sharing a directory between VMs is also done using Proper. Proper uses the `mount --bind` command to share a directory between VMs after checking that the action is authorized. Before Proper grants the request, Proper checks to see if the nest's slice has permission to mount a directory between slices. This permission has been granted to the nest's slice by PlanetLab's administrators. In addition to checking that the nest slice is authorized to share directories, the VM that the nest is sharing the directory with must authorize the sharing. This is performed by the VM creating a specially named file in the directory to authorize the sharing. Proper checks that this file exists in the directory and that the file contains the nest's slice name. Note that VServers did not require the VM to create a file to authorize the nest to share a directory because the nest runs in the VMM and so is treated as

trusted.

Sharing a file between VMs is performed by first sharing the VM's file system with the nest. Once the nest has a shared copy of the VM's file system, the nest can share a file with the VM by creating a hard link to the file on the VM's file system.

4.6 Client Tools

This section describes sharing from the perspective of the client tools. The section begins with a description of the communication between the VM and the nest, focusing on authentication (Section 4.6.1). Section 4.6.2 describes how the client tools use a nest transfer specialized plug-in to coordinate sharing with the nest to remove duplicate downloads. Section 4.6.3 discusses how the client tools use a package specialized plug-in to install packages that are shared between VMs by the nest.

4.6.1 Communication with the Nest

The client tools run in a VM and communicate with the nest to coordinate sharing. Communication is performed over a socket. The communication begins with mutual authentication of the VM and the nest. Once the VM has authenticated, the client tools can request files from the nest's cache as well as the installation of shared files from a package.

Authentication involves several steps by both the VM and the nest. The steps are:

1. A VM opens a socket to the nest, which is listening on a well-known port. The

nest is the only service allowed by the VMM to bind to that particular port. Thus the nest is authenticated to the VM.

2. The VM sends a message to the nest specifying the VM's name. This begins the process the nest uses to authenticate the VM.

3. The nest asks the VM to create a directory that the nest is allowed to share. The directory name is chosen by the nest using a secure random number generator.

4. The VM creates the directory. On the PlanetLab architecture the VM also authorizes the nest to share the directory by creating a specially named file in the directory with the nest slice name. In VServers the nest is in the VMM so does not need additional authorization to share the directory.

5. The nest attempts to share the directory onto the nest's file system.

6. If the nest can share the directory, then the nest has authenticated the VM because only the VM knew the directory name to create and the sharing will fail if the directory does not exist.

Once the VM and nest have authenticated each other, the client tools may request two actions: `sharecachedfile` and `sharepackage`. These actions are used by the VM to prevent duplicate downloads and provide secure sharing of packages.

- `sharecachedfile` Instructs the nest to share a file from its cache with the VM. If the file does not exist in the cache, the file is downloaded before sharing. The request includes a URL to the file and an optional secure hash of the file.

- `sharepackage` Requests that the nest share the files inside of a package with the VM. If the package has not been shared before, the nest prepares the package first by unpacking and protecting the files. The nest then shares the files in the package with the requesting VM.

4.6.2 Nest Transfer Specialized Plug-in

In addition to the specialized plug-ins that support BitTorrent, CoDeeN, Coral, HTTP, HTTPS, and FTP, there is a nest transfer specialized plug-in. This transfer specialized plug-in eliminates duplicate downloads by using a cache on the nest.

The nest transfer specialized plug-in provides an additional level of indirection so that the nest performs transfers on behalf of the VMs. The VM sends a `sharecachedfile` request to the nest for the files the VM wants downloaded. If the requested file is in the nest's cache, then the nest can share the file with the VM without downloading the file again. Otherwise, the nest will invoke a transfer specialized plug-in (such as BitTorrent, HTTP, etc) to retrieve the file. After the

nest retrieves the file, the nest adds the file to its cache and then shares the file with the requesting VM.

4.6.3 Shared Package Specialized Plug-in

A special package manager, `stork_nest_rpm`, is responsible for performing shared installation of RPM packages.

Installing a package that is shared by the nest is a two-phase process. In the first phase, `stork_nest_rpm` installs the package in an unshared manner. This allows the package to be installed atomically using the protections provided by `rpm`, including executing any install scripts. In the second phase, `stork_nest_rpm` sends the Stork nest a `sharepackage` request. This causes the nest to prepare the package for sharing and share the package with the VM. The unshared versions of package files that were installed by `rpm` are replaced by shared versions. Stork does not share configuration files because these files are often changed by a VM. Stork also examines files to make sure the files are identical prior to replacing a file with a hard link to the shared version. The files may not be identical if a package patches or compiles files as part of the installation process. There is a benign race condition in which the VM could modify the file after the nest checks and the nest could hard link over a file that differs. Since files in packages do not modify the underlying

documentation and executables, in practice this is not a problem.

Removal of packages that are installed using `stork_nest_rpm` requires no special processing. `stork_nest_rpm` calls `rpm` to uninstall packages in the same way as unshared package removals are handled. The `stork_nest_rpm` specialized plug-in uses the `rpm` tool to uninstall the package, which unlinks the package's files. The hard link count of the shared files is decremented, but is still nonzero because the shared files persist on the nest and in any other VMs that have installed the shared package.

4.7 Nest

The nest shares files with VMs on the same physical machine. Whenever a VM opens a connection with the nest, the nest first requires the VM to authenticate (previously described in Section 4.6.1). Once the VM is authenticated, the VM may request the `sharedcachedfile` and `sharepackage` operations both of which require the nest to share files using the share functionality manager (Section 4.7.1). `sharedcachedfile` is handled by the nest's cache (Section 4.7.2). `sharepackage` is first handled by the prepare functionality manager (Section 4.7.3) and then by sharing the unpacked files with the VM (Section 4.7.4).

4.7.1 Share Functionality Manager

The share functionality manager enables sharing between VMs on the same physical machine under the control of the nest. Specifically, a read-only file can be shared such that an individual VM cannot modify the file, although the file can be unlinked. The low level mechanisms used to build the sharing specialized plug-ins that are used by the sharing functionality manager were previously discussed in Section 4.5.

In Stork, sharing is provided via the share functionality manager that hides the details of sharing on different VM platforms. The sharing functionality manager is

used by the nest and provides four functions: `init_client`, `authenticate_client`, `share`, and `protect`. `init_client` is called when a VM opens a socket to the port the nest is listening on. `init_client` is used to initialize the per-VM state. `authenticate_client` is used by the nest to authenticate the VM with which it is communicating (Section 4.6.1). The `share` routine shares a file or directory between a VM and the nest. `protect` protects a file from modification by a VM.

There are different share specialized plug-ins for PlanetLab and VServers. These share specialized plug-ins perform architecture specific tasks to support sharing and protection on the platform (described in Sections 4.5.1 and 4.5.2).

4.7.2 Nest Cache

The purpose of the nest cache is to prevent the duplicate download of files. The nest caches the files it downloads and provides the files to VMs rather than VMs downloading the files multiple times. Some files that are downloaded by the client tools are referenced by secure hash (as described in Chapter 3) while the root repository metadata files are not. Files are handled differently by the nest cache depending on whether or not the secure hash is specified with the URL.

When the VM issues a `sharecachedfile` request, the nest cache handles the request differently if the VM provides the file's secure hash with the request. The

only file that a VM requests without a secure hash is the root repository metadata. The client tools do not know the secure hash of the root repository metadata since the root repository metadata changes periodically. A file requested without a secure hash is kept in the cache for 5 minutes. This means that each physical machine will download the root repository metadata from a repository at most once every 5 minutes.

In addition to root repository metadata requests by VMs, when PsEPR updates are pushed, the nest updates the nest's cache with the root repository metadata contained in the notification. Since the nest's cache is updated by PsEPR every 2 minutes, this means that if PsEPR is pushing notifications that are received by the nest, no VM will never need to download a root repository metadata file.

For all of the files other than the root repository metadata that are requested by a VM, the secure hash is known. This is true because when a metadata tarball is requested, the VM has the root repository metadata which contains the secure hash of the tarball. For packages, the VM has the package metadata which contains the secure hash of the package.

When the VM requests a file by secure hash, the file is provided from a cache that is indexed by secure hash. A file is added to the cache by creating a directory with the secure hash of the contents of the file and moving the file to the directory. This

allows the nest to check if the directory with a given secure hash exists to determine if an file exists in the cache. When a VM asks for the file with that secure hash, the file in that directory is shared into the VM's file system.

Files that are indexed by secure hash do not need to be expired from the cache for staleness reasons. This is true because the VM that requests the file knows the exact version of the file it desires (because this is the version signed by an entity). While it would be safe to remove some files from the nest cache (such as metadata tarballs that are out of date), in practice the nest's cache uses only a few MB of disk space a day so this has not been needed. Future work includes detecting and removing outdated files from the nest's cache.

4.7.3 Prepare Functionality Manager

The prepare functionality manager is used by the nest to prepare a package for sharing. When a VM issues a `sharepackage` request for a package the nest has not prepared before, the nest first uses the prepare functionality manager to unpack the package. If the nest has prepared the package before, the prepare functionality manager has no work to do and returns. In order to share a package, the nest must extract the files in the package. This extraction differs from package installation in that no installation scripts are run, the package database that tracks which packages

are installed is not updated, and the package files are not moved to the proper locations. This is because the package should not be installed in the nest. Instead, files are extracted to a directory in preparation for sharing with VMs.

Prepare specialized plug-ins only implement one function, `prepare`. The `prepare` function takes the name of a package and the destination directory where the package should be extracted.

The nest supports sharing RPM packages using the `stork_rpm_prepare` specialized plug-in. This works through the following steps:

1. The `stork_rpm_prepare` specialized plug-in checks if the package has already been prepared. If it has, then nothing needs to be done so the prepare specialized plug-in returns success.

2. If the package has not been prepared, then `stork_rpm_prepare` uses `rpm2cpio` to convert the RPM package into a cpio archive that is then extracted.

3. `stork_rpm_prepare` queries the `rpm` tool to determine which files are configuration files. Configuration files are often changed by a VM and so are not shared by the nest. The configuration files are removed from the prepare directory.

4. `stork_rpm_prepare` protects the files that were extracted so that the files

cannot be modified. This prevents a shared file from being modified by a VM.

4.7.4 Sharing a Prepared Package

After a package is unpacked in response to a VM's `sharepackage` request, the package's files must be shared with the VM. The package specialized plug-in (in Section 4.6.3) installed unshared versions of the package files. The nest replaces the unshared version of an installed package file with a hard link to a shared version.

The nest compares the VM's copy of a file with the nest's copy of a file prior to replacing a private copy with a hard link to the shared file. One instance in which the files may not be identical is if a package compiles files as part of the installation process. If the nest were to replace the file with the version the nest unpacked, this may break the package.

4.8 Walkthrough

Now that the mechanisms for sharing files have been described in detail, the examples presented in Section 4.2 are revisited. The steps that the VM and nest perform are described.

4.8.1 Duplicate Downloads

Summary: VM_A and VM_B on the same physical machine download the same package.

First, VM_A requests the package from the nest and the nest does not have the package in its cache.

1. VM_A authenticates with the nest.
2. VM_A sends a `sharecachedfile` request to the nest for the package. The request includes the secure hash of the package that the client tools in VM_A obtained from the package metadata.
3. The nest checks its cache to find there is no directory with the name of the secure hash.

4. The nest downloads the file from the URL provided in the `sharedcachedfile` request and adds the file to the cache.

5. The nest protects the file.

6. The nest shares the file with VM_A.

Now, VM_B requests the package from the nest and the package is in the nest cache.

1. VM_B authenticates with the nest.

2. VM_B sends a `sharecachedfile` request to the nest for the package. The request includes the secure hash of the package that the client tools in VM_B obtained from the package metadata.

3. The nest checks its cache to find a file with that secure hash exists in the nest's cache.

4. The nest shares the file with VM_B.

This example works the same if VM_A and VM_B download the file from different repositories. The example is the same for different repositories because the index of the nest's cache is done by secure hash. This means that the URL at which the file resides is unimportant unless the file is missing from the cache.

4.8.2 Duplicate Package Files

Summary: VM_A and VM_B on the same physical machine install the RPM package `foobar`. The package `foobar` has already been downloaded (as the previous example described).

First, VM_A installs the `foobar` package, which the nest has not previously prepared.

1. The client tools in VM_A uses the `stork_nest_rpm` package manager to install an unshared version of `foobar`.
2. VM_A authenticates with the nest.
3. VM_A sends a `sharepackage` request to the nest for the package.
4. The nest checks and determines it has not previously prepared the package.
5. The nest uses the prepare functionality manager to unpack the package.
6. The prepare functionality manager uses the share functionality manager to protect all of the files in the package.
7. The nest uses the share functionality manager to replace the files that are the same on VM_A and the prepared directory with a hard link to the file in the

prepared directory. This means that VM_A now has a shared version of those files.

8. The nest informs VM_A that the package is now shared.

Now, VM_B installs the `foobar` package.

1. The client tools in VM_B use the `stork_nest_rpm` package manager to install an unshared version of `foobar`.
2. VM_B authenticates with the nest.
3. VM_B sends a `sharepackage` request to the nest for the package.
4. The nest checks and determines it has prepared the package previously.
5. The nest uses the share functionality manager to replace the files that are the same on VM_B and the prepared directory with a hard link to the file in the prepared directory. This means that VM_B now has a shared version of those files.
6. The nest informs VM_B that the package is now shared.

4.9 Sharing Trade offs

Now that the client tools and nest have been described in detail, it is important to revisit the different nest architectures to understand their trade offs. The nest architectures for PlanetLab and VServers present different design decisions. The methods for sharing in VServers and PlanetLab discussed in Sections 4.5.1 and 4.5.2 are significantly different and each have their strengths and weaknesses related to security and implementation complexity.

A compromise of the nest on a physical machine may compromise the VMs the nest serves. This is true because the nest has the ability to access the VMs' file systems for the purpose of replacing duplicate files with hard links to shared files. However, the underlying architecture affects the result of a nest compromise.

PlanetLab isolates the nest VM from other VMs using Proper. Proper allows the nest to access another VM's file system only when authorized by the VM. This means that a compromise of the nest does not imply a compromise of all VMs on the physical machine. Only the VMs that use Stork are at risk.

On VServers, since the nest runs in the VMM the nest becomes part of the trusted computing base. This means a compromise of the nest results in an attacker having control of all of the VMs on the physical machine. This implies that if all else were

equal, the sharing architecture of Stork on the PlanetLab platform would provide greater security than the VServers architecture.

However, the isolation and authorization provided by Proper on PlanetLab is not without cost. The interactions with Proper on PlanetLab increase the complexity of the PlanetLab sharing specialized plug-in substantially. Not only does the nest need to communicate with Proper to perform actions, but the nest must act intelligently when Proper fails to perform an operation. Given the limited information available to the nest when running in a VM, acting intelligently on limited information is difficult and greatly increases the amount of error handling code needed. The result is that the PlanetLab sharing specialized plug-in is over twice the size (281 LOC) as the VServers sharing specialized plug-in (115 LOC), not including the size of Proper (5752 LOC) [61]. The implementation effort involved in Proper is also substantial. In comparison, the nest is less than one quarter the size of Proper (1404 unique LOC with 8736 LOC of libraries) if libraries used by other components of Stork are not included. This means that the implementation effort to provide sharing through the Stork nest on PlanetLab is substantially greater than in the VServers environment.

The architecture that provides the best security depends on the scenario. If all VMs on a physical machine use the nest, there is little incentive to place the nest in a VM because either way a compromise of the nest results in the compromise of all VMs. In fact, since the implementation effort to run the nest in a VM is greater

than running the nest in the VMM, it is arguably less secure to have the nest in a VM because more lines of code leave more opportunities for bugs. If only a few VMs on a physical machine use Stork, putting the nest in a VM makes a great deal of sense because a compromise of the Stork nest will not compromise the VMs that do not use Stork.

4.10 Related Work

Most VMMs focus on providing isolation between VMs, not sharing. However different techniques have been devised to mitigate the network bandwidth consumed by downloading duplicate data (Section 4.10.1), as well as the disk (Section 4.10.2) and memory (Section 4.10.3) inefficiencies of unshared packages.

Overall, Stork's technique for reducing duplication cannot be replaced by a single component. This is because the techniques described that prevent downloading duplicate data do not provide disk or memory savings and vice versa. However, there are also differences between how other work reduces duplication versus how the Stork nest reduces duplication.

4.10.1 Network Bandwidth

A common technique to mitigate the network costs of duplicate data retrieval is to use a proxy server [19, 65, 74, 87]. Proxy servers minimize the load on the server providing the data and also increase the performance of the clients. If the proxy server is not on the same physical machine, the data still must be transferred multiple times over the network. The Stork nest provides the data to the VMs without incurring network traffic much like each physical machine running its own

proxy server for packages. However, running a proxy server on each physical machine server will not prevent the same file from being downloaded multiple times from different repositories (as described in the example in Section 4.2.1). The Stork nest prevents duplicate downloads regardless of source.

4.10.2 Disk

A good deal of research has gone into preventing duplicate data from consuming additional disk space. For example, many file systems use copy-on-write techniques [14, 20, 34, 39, 40, 79] which allow data to be shared but copied if modified. This allows different "snapshots" of a file system to be taken where the unchanged areas will be shared amongst the "snapshots". However, this does not combine identical files that were written at different locations (as would happen with multiple VMs downloading the same package). This means that this research would fail to provide any gains in package management efficiency. However, the Stork nest could leverage this functionality to share the files inside of packages without using protection mechanisms. Additionally, the nest could leverage this functionality to share configuration files between VMs.

Some filesystem tools [11] and VMMs [43, 45] share files that have already been created on a file system. They unify common files or blocks to reduce the disk space

required. This unification happens after the package has been installed; each VM must download and install the package separately only to have a background process detect and replace files with hard links to the shared version. Stork avoids duplicate downloads and does not require a background process to discover duplicate copies of package files.

Another technique for reducing the amount of storage space consumed by identical components detects duplicate files and combines the duplicate files as they are written [64]. This is typically done by using a hash of the file blocks to quickly detect duplicates. Stork avoids the overhead of needing to check file blocks for duplicates on insertion and avoids the need to download the block multiple times in the first place. However, block-based de-duplication techniques have greater potential for sharing than techniques that operate on the whole file.

4.10.3 Memory

There are many proposals that try to reduce the memory overhead of duplicate memory pages. Disco [14] implements copy-on-write memory sharing between VMs which allows not only a process' memory pages to be shared but also allows duplicate buffer cache pages to be shared. The sharing of package executables provided by the nest in Stork is much less effective than Disco, but at a much lower

cost.

Stork allows VMs to share the memory used by shared applications and libraries. VMware ESX Server [84] also allows VMs to share memory, but does so based on page content. A background process scans memory looking for multiple copies of the same page. Any duplicate copies are eliminated by replacing the copies with a single copy-on-write page. This allows for more potential sharing than Stork, as any identical pages can be shared, but induces overhead when scanning pages for duplicates. Also Stork prevents memory duplication in the first place instead of having processes create duplicate pages only to have the duplicate pages culled.

4.11 Conclusion

This chapter described how Stork uses sharing to make package management more efficient in VM environments. Some examples that describe sharing in VM environments were provided (Section 4.2) and later examined in detail (Section 4.8). Then the benefits of sharing were analyzed to show that sharing can greatly reduce the network bandwidth, disk space, and memory used in VM environments (Section 4.3). Following that, an overview of sharing in Stork was provided in Section 4.4. Different mechanisms for sharing in different architectures was provided in Section 4.5. Then the VM (Section 4.6) and nest (Section 4.7) actions to facilitate sharing were described. Section 4.9 provided a discussion of the different locations where the nest can reside and the implications this has on security and ease of implementation. Section 4.10 discussed related work that reduces duplicate downloads or mitigates the inefficiencies of VMs that have an unshared package on the same physical machine.

Chapter 5

Conclusion

Package managers are an important tool for managing software on modern computers. However, care must be taken in the design of a package manager. There are two areas that are not handled well by existing package managers: security and VM environments. These areas are important because of the increasing number of security threats and the increasing popularity of VM environments. This work demonstrates how to provide secure package management and how to efficiently operate in VM environments using sharing.

By identifying and classifying flaws in existing package managers, areas of attack are identified. Unfortunately architectural flaws are the reason for vulnerability to attack in many cases. The security architecture of Stork was designed to protect against these attacks. As a result Stork protects clients against the attacks that are effective on other package managers.

Virtual machine environments present an efficiency challenge for package managers. The VMM software isolates different VMs on the same physical machine which

leads existing package managers to waste network bandwidth, disk space, and memory because of duplication. Stork presents an architecture that uses sharing to remove duplication in downloaded files and installed package files. Sharing downloaded files saves network bandwidth for downloaded packages and package metadata. Sharing installed package files saves disk space and memory.

Stork is a real system for package management that is secure and provides efficiency in VM environments. Stork downloads files using multiple transfer protocols, supports different package formats, does dependency resolution, and has simple package-based configuration management facilities. Stork provides a complete solution to the problem of package management and has clients that are VMs as well clients that are normal systems. Stork is in everyday use — the main Stork repository serves a package about once every 20 seconds. Stork has also been used as a component when building other services. Stork is used daily by administrators at two dozen institutions. The package-based configuration management utilities manage VMs on hundreds of failure prone, globally distributed, physical machines. Stork has been used for 4 years and has managed half a million VM instantiations.

5.1 Future Work

There are several optimizations that are being considered for future work:

The nest keeps a cache of files downloaded and provides files from the cache to VMs on the same physical machine. The files that are downloaded without a secure hash are flushed after a short time period. However, the files that are downloaded by secure hash are not flushed. While it would be safe to remove some files that are downloaded by secure hash from the nest cache (such as metadata tarballs that are out of date), in practice the nest's cache uses less than 10MB of disk space a day so this has not been needed. Future work would focus on detecting and removing outdated files from the nest's cache.

There are subtle issues that the nest raises with respect to resource accounting. Given that the nest is performing a transfer on the behalf of a VM, it seems logical that the resources the nest consumes when performing that transfer should be attributed to the VM. However, this is a non-trivial alteration to the VMM of existing VM environments and so is left to future work.

The package-based configuration management tools provide an effective mechanism for changing the configuration of a client. However, the package-based configuration management tools do not provide feedback to the administrator about the actions

that occurred. Adding feedback about the state of the administrator's clients would greatly increase the usefulness of Stork's package-based configuration management tools.

Appendix A

Definitions

This chapter describes the terminology used throughout the remainder of the document.

administrator An administrator is a party that controls a computer. The predominant use of the term administrator in this dissertation refers to the administrator of a client. When used in a different context, administrator is preceded by a qualifier that indicates what the administrator controls. For example, a package repository is controlled by a repository administrator.

attacker An attacker is a party that attempts to subvert security mechanisms, perhaps for malicious reasons.

client A client is a generic term for a system or VM that performs package management.

client tools The client tools component of Stork runs on a client and manages the packages installed on the client. The client tools are comprised of `stork`, `pacman`, and `stork_receive_update`.

dependency A dependency is functionality that a package must have to operate. Packages may have a dependency on another package, a virtual package, or a specific file.

dependency resolver A dependency resolver is the component of a package manager that administrators normally interact with. A dependency resolver performs dependency resolution and downloads packages. The dependency resolver instructs a package installer to install, remove, or update packages.

dependency resolution Dependency resolution is a process that, when given a package, finds a collection of packages to install to meet all dependencies. After installation, there are no packages installed on the client that have unmet dependencies.

entity An entity possesses a public key and private key and makes decisions about the trustworthiness of packages and metadata. Administrators, developers, projects, repositories, and distributions are examples of entities.

group A group is a logical association of multiple clients defined by the administrator of those clients for the purpose of simplifying package-based configuration management. Groups are defined in the administrator's `groups.pacman` file.

`groups.pacman` The `groups.pacman` file defines which clients belong to which groups for the purpose of package-based configuration management.

functionality manager A functionality manager is an abstraction used in different components of Stork to provide extensibility. A functionality manager hides the complexity of how something is done. A functionality manager handles a general type of operation (such as a file transfer) and uses specialized plug-ins to perform operations in a specific manner (such as downloading files using a specific protocol).

install Installing a package is performed by unpacking the files in the package on a client so that the software can be used. If a version of the package is already installed, this operation does nothing.

master configuration file The administrator's master configuration file contains the configuration options for the client tools component of Stork.

metadata Metadata is a generic term for information other than packages that is downloaded from a repository. In Stork, metadata includes `packages.pacman` files, `groups.pacman` files, TP files, package metadata, the root repository metadata, and master configuration files.

mirror A mirror is a copy of a repository usually created for load balancing or failure handling. A mirror typically does not intentionally deviate from the repository it copies and so does not sign metadata or packages itself.

nest The nest component of Stork coordinates sharing of packages between different

VMs on the same physical machine.

package A package is software that is bundled with information that describes the software. The information bundled with a package is called package metadata.

package action A package action is an operation on a software package, typically install, update, or remove.

package-based configuration management The package-based configuration management component of Stork allows an administrator to control clients collectively without logging into each client separately.

package database A package database stores information about the packages currently installed on the client. A package database is managed by the package installer and is used to assist in performing package actions on a client.

package instructions An administrator can create package instructions using the package-based configuration management tools. These instructions are stored in the `groups.pacman` and `packages.pacman` files. A client will perform package actions based upon the administrator's package instructions.

package installer A package installer is a program that installs, updates, or removes a package from a client. A package installer tracks installed packages using a package database.

package manager A package manager is a program that manages the installation, update, and removal of software on a client. A package manager is typically split into two subcomponents: a package installer and a dependency resolver. The package manager is commonly referred to by the name of the dependency resolver since most dependency resolvers only support one package installer and administrators interact with the dependency resolver component.

package metadata Package metadata is information about a package. Different package formats have different information. The package metadata typically describes the package, lists the dependencies of the package, and lists what dependencies the package provides.

`packages.pacman` The administrator's `packages.pacman` file defines the package-based configuration management actions that the clients in a group should enact.

physical machine A physical machine is the hardware comprising a computer. Different VMs can run on a single physical machine.

provides The dependencies that a package can be used to resolve are the dependencies that the package provides. The dependencies a package provides are listed in the package metadata for the package.

remove Removing a package deletes the package files from the client and updates

the package database to indicate the package is no longer installed.

repository A repository (or package repository) is a web server that serves packages and package metadata instead of web pages. In Stork, a package repository also validates packages and metadata uploaded by entities.

root repository metadata The root repository metadata is the file on a repository that describes the packages and metadata available on the repository.

slice A slice is a collection of VMs and related resources on PlanetLab with common administration.

specialized plug-in A specialized plug-in supports a specific way of performing an operation. Specialized plug-ins are used to support different ways of performing an action such as different communication protocols and different package formats. Specialized plug-ins are used by functionality managers.

system A system is a computer that does not use virtual machine technologies. In other words, a system is a classic desktop computer that is managed by a package manager. There is one physical machine for each system.

tag A tag is a mechanism used in TP files in Stork to specify a preference when multiple packages fulfill a dependency.

trusted packages (TP) file A trusted packages file is used to specify trust in packages and entities.

update As opposed to install, update changes an existing package to the preferred version (usually the most recent) by removing the existing version and installing the preferred version. If the package is not installed, this operation installs the package.

virtual machine (VM) A virtual machine is an operating system virtualized environment as is provided by Xen, VMWare, PlanetLab, or VServers. To the administrator of the VM, this is conceptually similar to a system.

virtual machine monitor (VMM) The software that manages and mediates the actions of different VMs on the same physical machine is called a virtual machine monitor (VMM). The VMM has complete control over the VMs on the physical machine (analogous to root access on a multi-user operating system).

virtual dependency A virtual dependency is a dependency on a virtual package.

virtual package A virtual package represents functionality that is independent of the package that provides the functionality. A virtual package is commonly used when a package wants to specify a dependency, but that the dependency is not on a specific piece of software but instead a type of functionality. For example, there is a webbrowser virtual package that is provided by every package that is a web browser. A package that needs a web browser to operate can depend on the webbrowser virtual package.

References

[1] Adobe - Adobe Acrobat Family. http://www.adobe.com/products/acrobat/?ogn=EN_US-gntray_prod_acrobat_family_home.

[2] Adium - Download. http://www.adiumx.com/.

[3] Albrecht, J., Tuttle, C., Snoeren, A. C., and Vahdat, A. PlanetLab application management using Plush. *SIGOPS Oper. Syst. Rev. 40*, 1 (2006), 33–40.

[4] PlanetLab Application Manager. http://appmanager.berkeley.intel-research.net/.

[5] Debian APT tool ported to RedHat Linux. http://www.apt-get.org/.

[6] APT-RPM. http://apt-rpm.org/.

[7] Arch Linux (Don't Panic) Installation Guide. http://www.archlinux.org/static/docs/arch-install-guide.txt.

[8] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the Art of Virtualization. In *Proc. 19th SOSP* (Lake George, NY, Oct 2003).

[9] Bellissimo, A., Burgess, J., and Fu, K. Secure Software Updates: Disappointments and New Challenges. In *1st USENIX Workshop on Hot Topics in Security* (Vancouver, Canada, Jul 2006), pp. 37–43.

[10] Berman, F., Chien, A., Cooper, K., Dongarra, J., Foster, I., Gannon, D., Johnsson, L., Kennedy, K., Kesselman, C., Mellor-Crummey, J., Reed, D., Torczon, L., and Wolski, R. The GrADS Project: Software support for high-level Grid application development. *The International Journal of High Performance Computing Applications 15*, 4 (2001), 327–344.

[11] Bolosky, W. J., Corbin, S., Goebel, D., and Douceur, J. R. Single Instance Storage in Windows 2000. In *Proc. 4th USENIX Windows Sys. Symp.* (Seattle, WA, Aug 2000), pp. 13–24.

[12] Brett, P., Knauerhase, R., Bowman, M., Adams, R., Nataraj, A., Sedayao, J., and Spindel, M. A shared global event propagation system to enable next generation distributed services. In *Proc. of the 1st Workshop on Real, Large Distributed Systems* (San Francisco, CA, Dec 2004).

[13] Installing Applications: Packages and Ports. `http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/ports.html`.

[14] Bugnion, E., Devine, S., Govil, K., and Rosenblum, M. Disco: running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems 15*, 4 (Nov 1997), 412–447.

[15] Cappos, J., Baker, S., Plichta, J., Nyugen, D., Hardies, J., Borgard, M., Johnston, J., and Hartman, J. Stork: Package Management for Distributed VM Environments. In *Proc. 21th Systems Administration Conference (LISA '07)* (Dallas, 2007).

[16] Cappos, J., and Hartman, J. Why It Is Hard to Build a Long Running Service on Planetlab. In *Proc. of the 2nd Workshop on Real, Large Distributed Systems* (San Francisco, CA, Dec 2005).

[17] CERT. `http://www.cert.org/`.

[18] Cfengine - an adaptive system configuration management engine. `http://www.cfengine.org/`.

[19] Chankhunthod, A., Danzig, P. B., Neerdaels, C., Schwartz, M. F., and Worrell, K. J. A Hierarchical Internet Object Cache. In *USENIX Annual Technical Conference* (1996), pp. 153–164.

[20] Chutani, S., Anderson, O. T., Kazar, M. L., Leverett, B. W., Mason, W. A., and Sidebotham, R. N. The Episode file system. In *Proceedings*

of the USENIX Winter 1992 Technical Conference (San Fransisco, CA, USA, 1992), pp. 43–60.

[21] Introduction to Code Signing. `http://msdn2.microsoft.com/en-us/library/ms537361.aspx`.

[22] Cohen, B. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems* (2003).

[23] Debian Developer's Reference. `http://www.debian.org/doc/packaging-manuals/developers-reference/`.

[24] debsigs - What is debsigs. `http://linux.about.com/cs/linux101/g/debsigs.htm`.

[25] DistroWatch.com: Editorial: How Popular is a Distribution? `http://distrowatch.com/weekly.php?issue=20070827#feature`.

[26] Debian – dpkg. `http://packages.debian.org/stable/base/dpkg`.

[27] man dpkg-sig. `http://pwet.fr/man/linux/commandes/dpkg_sig`.

[28] dsniff. `http://monkey.org/~dugsong/dsniff/`.

[29] 'Gentoo Linux Documentation – Ebuild HOWTO'. `http://www.gentoo.org/proj/en/devrel/handbook/handbook.xml?part=2&chap=1`.

[30] Eugster, P. T., Felber, P., Guerraoui, R., and Kermarrec, A.-M. The Many Faces of Publish/Subscribe. *ACM Computing Surveys 35*, 2 (Jun 2003), 114–131.

[31] Infrastructure/Mirroring – Fedora Project Wiki. `http://fedoraproject.org/wiki/Infrastructure/Mirroring`.

[32] Firefox web browser — Faster, more secure, & customizable. `http://www.mozilla.com/en-US/firefox/`.

[33] Freedman, M. J., Freudenthal, E., and Mazières, D. Democratizing content publication with Coral. In *Proc. 1st NSDI* (San Francisco, CA, Mar. 2004).

[34] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. In *Proc. 19th SOSP* (Lake George, NY, Oct 2003).

[35] Gkantsidis, C., Karagiannis, T., and VojnoviC, M. Planet scale software updates. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2006), ACM, pp. 423–434.

[36] Godber, A., and Dasgupta, P. Countering rogues in wireless networks. In *2003 International Conference on Parallel Processing Workshops* (Oct 2003).

[37] Goldsack, P., Guijarro, J., Lain, A., Mecheneau, G., Murray, P., and Toft, P. Smartfrog: Configuration and automatic ignition of distributed applications. Tech. rep., HP, 2003.

[38] GovCertUK. http://www.govcertuk.gov.uk/.

[39] Hitz, D., Lau, J., and Malcolm, M. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference* (San Fransisco, CA, USA, 17–21 1994), pp. 235–246.

[40] Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst. 6*, 1 (1988), 51–81.

[41] InstallShield – Installation Tool. http://www.macrovision.com/products/installation/installshield.htm.

[42] MMi Public Service Announcement – Malicious Installer Source Warning. http://www.modmyifone.com/forums/showthread.php?t=24323.

[43] Kamp, P.-H., and Watson, R. N. M. Jails: Confining the Omnipotent Root. In *Proc. 2nd Int. SANE Conf.* (Maastricht, The Netherlands, May 2000).

[44] The KPackage Handbook. http://docs.kde.org/development/en/kdeadmin/kpackage/.

[45] Linux VServers Project. http://linux-vserver.org/.

[46] Linux/Lupper.worm. http://vil.nai.com/vil/content/v_136821.htm.

[47] Apple – Software Update. http://www.apple.com/softwareupdate/.

[48] Mazières, D., Kaminsky, M., Kaashoek, M. F., and Witchel, E. Separating key management from file system security. In *Proc. 17th SOSP* (Kiawah Island Resort, SC, Dec 1999), pp. 124–139.

[49] Mazières, D., and Shasha, D. Building secure file systems out of Byzantine storage. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing* (New York, NY, USA, 2002), ACM, pp. 108–117.

[50] Office Online Home Page - Microsoft Office Online. http://office.microsoft.com/en-us/default.aspx.

[51] Muir, S., Peterson, L., Fiuczynski, M., Cappos, J., and Hartman, J. Proper: Privileged Operations in a Virtualised System Environment. In *Proc. USENIX '05* (Anaheim, CA, Apr 2005).

[52] Netcraft: Strong growth for Debian. http://news.netcraft.com/archives/2005/12/05/strong_growth_for_debian.html.

[53] Oppenheim, K., and McCormick, P. Deployme: Tellme's Package Management and Deployment System. In *Proc. 14th Systems Administration Conference (LISA '00)* (New Orleans, LA, Dec 2000), pp. 187–196.

[54] Re: Differences of Debian vs. the Other Guys. `http://lists.debian.org/debian-devel/1998/06/msg00128.html`.

[55] Comparing Linux/UNIX Binary Package Formats. `http://kitenet.net/~joey/pkg-comp/`.

[56] Package Management System - Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Package_manager`.

[57] Park, K., and Pai, V. S. Deploying Large File Transfer on an HTTP Content Distribution Network. In *Proc. of the 1st Workshop on Real, Large Distributed Systems* (San Francisco, CA, Dec 2004).

[58] Park, K., and Pai, V. S. Scale and Performance in the CoBlitz Large-File Distribution Service. In *Proc. 3rd NSDI* (San Jose, CA, May 2005).

[59] Park, K., and Pai, V. S. CoMon: A Mostly-Scalable Monitoring System for PlanetLab. *Operating Systems Review Special Issue on PlanetLab* (Jan 2006).

[60] Percival, C. An Automated Binary Security Update System for FreeBSD. pp. 29–34.

[61] Peterson, L., Bavier, A., Fiuczynski, M. E., and Muir, S. Experiences building PlanetLab. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 351–366.

[62] PlanetLab. `http://www.planet-lab.org`.

[63] 'Gentoo Linux Documentation – A Portage Introduction'. `http://www.gentoo.org/doc/en/handbook/handbook-x86.xml?part=2&chap=1`.

[64] Quinlan, S., and Dorward, S. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies* (Monterey,CA, 2002).

[65] Rabinovich, M., Chase, J., and Gadde, S. Not all hits are created equal: cooperative proxy caching over a wide-area network. *Comput. Netw. ISDN Syst. 30*, 22-23 (1998), 2253–2259.

[66] Symantec Security Response - Linux.Ramen.Worm. `http://service1.symantec.com/sarc/sarc.nsf/html/Linux.Ramen.Worm.html`.

[67] Red Hat Network - About RHN. `https://rhn.redhat.com/help/about.pxt`.

[68] RPM Package Manager. `http://www.rpm.org/`.

[69] Samuel, J., Plichta, J., and Cappos, J. Centralized Package Management Using Stork. *;login:* (Feb 2008), 25–31.

[70] SecureApt - Debian Wiki. http://wiki.debian.org/SecureApt.

[71] Skype - free Internet calls and cheap calls to phones and mobiles. http://www.skype.com/useskype/.

[72] Slackware Package Management. http://www.slacksite.com/slackware/packages.html.

[73] Installation (computer programs) - Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Installer.

[74] squid : Optimising Web Delivery. http://www.squid-cache.org/.

[75] SquirrelMail Repository Poisoned with Critical flaw. http://www.beskerming.com/commentary/2007/12/19/313/SquirrelMail_Repository_Poisoned_with_Critical_flaw.

[76] Stork. http://www.cs.arizona.edu/stork.

[77] StuffIt InstallerMaker for Macintosh. http://my.smithmicro.com/mac/stuffitinstallermaker/index.html?im-index.html.

[78] Synaptic Package Manager - Home. http://www.nongnu.org/synaptic/.

[79] Thekkath, C. A., Mann, T., and Lee, E. K. Frangipani: a scalable distributed file system. In *SOSP '97: Proceedings of the sixteenth ACM sympo-*

sium on Operating systems principles (New York, NY, USA, 1997), ACM Press, pp. 224–237.

[80] RSA Alert: New Universal Man-in-the-Middle Phishing Kit Discovered. `http://www.rsa.com/press_release.aspx?id=7667`.

[81] URPMI. `http://www.urpmi.org/`.

[82] Venema, W. Murphy's law and computer security. In *SSYM'96: Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography* (Berkeley, CA, USA, 1996), USENIX Association, pp. 19–19.

[83] VMWare Workstation. `http://www.vmware.com/`.

[84] Waldspurger, C. A. Memory resource management in VMware ESX server. *Operating Systems Review 36* (2002), 181–194.

[85] Wang, L., Park, K., Pang, R., Pai, V., and Peterson, L. Reliability and Security in the CoDeeN Content Distribution Network. In *Proc. USENIX '02* (San Francisco, CA, Aug 2002).

[86] Windows Update. `http://update.windows.com/`.

[87] WinProxy. `http://www.winproxy.com/index.asp`.

[88] Software Packaging and Installation Authoring Tools – Altris, Inc. `http://www.wise.com/`.

[89] Wurster, G., and van Oorschot, P. Self-Signed Executables: Restricting Replacement of Program Binaries by Malware. In *2nd USENIX Workshop on Hot Topics in Security* (Boston, MA, Aug 2007).

[90] YaST - openSuSE. `http://en.opensuse.org/YaST`.

[91] Yuan, L., Kant, K., Mohapatra, P., and Chuah, C.-N. DoX: A Peer-to-Peer Antidote for DNS Cache Poisoning Attacks. In *2006 IEEE International Conference on Communications* (Jun 2006).

[92] Yum: Yellow Dog Updater Modified. `http://linux.duke.edu/projects/yum/`.