Secure Dependency Enforcement in Package Management Systems

Luigi Catuogno, Clemente Galdi, and Giuseppe Persiano

Abstract—Package management systems play an essential role in pursuing systems dependability by ensuring that software is correctly installed and kept up-to-date according to vendor-defined installation policies. Circumventing such policies could make the system unhealthy and insecure and can constitute a serious security threat. In many application scenarios, e.g., distribution of commercial software, the confidentiality of the software must be guaranteed against non-authorized players. In some cases, the installation policy itself is considered a sensitive information, e.g., when it reveals required hardware in military contexts. In this paper we address the problem of strongly enforcing software dependencies in package management systems, to prevent that a malicious user forces the system to install any package despite its requirements are not completely fulfilled. The enforcement is *strong* in the sense that the encrypted software package cannot be even decrypted if the dependencies are not satisfied. Once a new package is decrypted and installed, our protocol *non-interactively* updates the key material on the target device. This key update will allow the decryption of further packages that depend on the newly installed one. We further present "policy-hiding" variants of our protocol. Finally we provide an experimental evaluation of the system performance.

Index Terms—Package Management Systems; Secure Software Update; Dependency Enforcement.

1 Introduction

SOFTWARE deployment is a critical activity within the life cycle of computer application systems. The tasks of distribution, installation, and update of software components require to be carefully designed and managed in order to keep the whole system safe throughout their execution. A fault occurring during any of the above tasks might prejudice the system integrity and trustworthiness.

Distribution facilities are required to not disseminate malformed or corrupted components as well as deployment agents are required to install/update a component on the system only if it fulfils the requirements posed by its vendor. For example, an application might come with the list of those required libraries that should be already present on the system, as well as the system could enforce any kind of software updates policy concerning of version, freshness, provenance and so on. Generally, such properties and requirements are referred to as *dependencies*.

Dependencies link different system components so that it is possible to identify which are the components that are affected by the installation or update operations. For this reason, ensuring integrity and coherence of dependencies through deployment tasks is a crucial activity which is pursued at every step of the software lifecycle, as any failure during such a process may have disastrous consequences. Indeed, recent studies [2] identify dependencies breaking (during component deployment tasks) as a major cause of systems malfunctioning.

In many scenarios, the software itself might be considered sensitive information, e.g., in military applications. In these contexts, it is crucial to preserve software confidentiality w.r.t. users that are *not* allowed to install it. Furthermore, it might be the case that the confidentiality of the dependencies themselves should be protected as they may leak information about the software, e.g., the requirement for the driver of a *weapon* for the drone reveals the nature of the mission.

The purpose of the system we present is twofold.

Legitimate user security: Our system guarantees authenticity, integrity and freshness for legitimate users. This means that a given software is installed on an untampered device if and only if (a) It is authentic; (b) unmodified; and (c) all the required dependencies are properly satisfied.

Prevent unauthorized installations: A malicious user that is able to collect 'files' from different devices will not be able to install a given software even if the union of all collected files satisfies the required dependencies.

We consider software deployment within modern component based applications, including operating systems where deployment is mainly performed through either public or private networks. Deployable units, such as "packages", "updates", "patches", are *produced* by a *deployment server*, *published* by a set of *mirror servers* and then *downloaded and installed/applied* by *client* machines either automatically or manually. This raises new issues related to security aspects. Indeed, adversaries could threaten their targets by exploiting any vulnerability throughout the whole software distribution chain. In this way, a fraudulent deployment/mirror server, by impersonating a legitimate one, could distribute malformed/counterfeit packages. Similarly,

- *L. Catuogno is with Dipartimento di Informatica, Università degli Studi di Salerno, Salerno (SA), Italy.*
- *C. Galdi is with Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione, Università degli Studi di Napoli "Federico II", Napoli (NA), Italy*
- *G. Persiano is with Dipartimento di Scienze Aziendali - Management & Innovation Systems, Università di Salerno, Salerno (SA), Italy. Work done while visiting Google.*

This work has been partially supported by the Italian POR projects "WISCH, Work Into Shaping Campania's Home" *and by* "Piattaforma di Mobilità Intelligente basata su sistema Multi-Agente".
A preliminary version of this paper appears in [1].

by having the control of any network segment, an adversary could interfere with the deployment protocols, propagating misleading information and messages in order to delay or avoid the installation/update of certain components, so that the target systems remain in or pass to an unsafe or faulty state.

Exploiting software vulnerabilities or network ones have been widely addressed in the literature though, few solutions face both aspects as a whole. Indeed, in many cases, security features are designed as wrappers for existing deployment facilities, so that the formers encompass client-server mutual authentication and communication confidentiality, while the latters have in charge the verification of satisfiability and consistence of dependencies. However, as noticed above, both aspects influence each other as, on one hand, malicious protocol deviations may trigger unsafe deployment actions and on the other hand, wrong installation/update procedure may lead to the appearance of new security breaches.

In this paper we present a comprehensive mechanism for security and reliability in software deployment, focusing the attention on the scenario depicted by "Package Management Systems" as a special case of software deployment systems.

We consider the case in which *multiple independent* software vendors provide encrypted software *packages* along with their *installation policies*. Our system guarantees on one hand the confidentiality, integrity, freshness and authenticity properties of the packages. On the other hand, it allows a *strong enforcement* of dependencies in the sense that if some required dependency has not been installed using our system the package cannot even be decrypted and, consequently, cannot be installed. Each new package provides a number of *features*, each associated to a *random feature key* that are used to encrypt every future package that depends on the current. The same feature can be provided by multiple vendors. In this case, each vendor will provide its random feature key and we assume the existence of a naming strategy that allows the unique identification of features. In this respect, this protocol provides a *non-interactive* procedure that allows each device to update the set of feature keys associated to the installed packages.

We then present an extension of our protocol in which the installation policy can be hidden from the user. In other words, if the user is missing some of the prerequisite for the installation, she cannot decrypt the package and, at the same time, she is not able to say which package she is missing.

Our protocols constitute a sharp improvement with respect to previous solutions in which either package security and dependencies are *managed separately*, or target systems are essentially *static*, or software vendors have to cooperate to generate packages keys, i.e., software vendors *cannot be independent*, or the user and the vendor have to run an interactive protocol that is either *inefficient* or *leaks the device profile*, i.e., the list of installed packages.

2 Related Works

One of the first papers addressing security issues in software deployment via Internet is [3], where the author suggests to introduce a trusted third party (*certifier*) who creates digital certificates for software vendors/developers (*issuers*). Each time a new package is released, the issuer provides the certifier with a signed certificate for the software that is then signed by the certifier. The user who wishes to install the package (*consumer*) is enabled to verify the authenticity of the software by verifying the certifier's signature.

Such a solution is extremely simple and flexible and it is used as a baseline for modern package managers. Nevertheless, it has been shown in [4], [5], that it is potentially prone to a number of vulnerabilities due to the improper use of cryptographic primitives. In [6] the authors show that several major package management systems are vulnerable to *man-in-the-middle* attacks. In particular, [5] highlights the threat of malicious mirrors and third-party components in the distribution network (*e.g.*, HTTP proxies).

In [7], authors emphasises the importance for a package management system to feature an effective content authentication mechanism. A secure software distribution infrastructure, leveraging run-time application integrity verification (*code signing*), is envised in [8]. Microsoft Windows systems introduced the Authenticode [9] technology to perform package authentication by means of code signing. Similarly, the Android OS features a code signing mechanism partially inherited from the Java framework [10]. However in [7] authors warn about the weak or non-existent resilience of some of such solutions to possible key disclosure and the lack of "trust revocation" mechanisms.

Authors in [11] address the issue of confidentiality of software updates where package distribution is carried out through partially or totally untrusted multiple mirrors. The paper introduces a scheme to manage package downloads without releasing information about the requested software. The scheme implements a Private Information Retrieval (PIR) [12] scheme. However, the solution presented in this paper has the strong requirement that each software download has to be executed by concurrent communication with multiple repositories.

Several works highlights that package dependencies remains a major matter of concern [13]. The RedHat's Package Manager (RPM) [14], which can somehow be regarded as the archetype of the major current package management systems, has suffered from serious problems related to a weak mechanism to validate and enforce dependencies among different packages [15]. In particular, a relation between dependence inconsistency and security vulnerabilities has been put forth in [16].

In [17] the authors face the problem of achieving software confidentiality in a context in which the system provides updates to millions of devices through a cache-enabled network. In the envisioned scenario, client devices are partitioned in *classes* according to their own distinguishing set of *device attributes* (*e.g.*, processor, OS, . . .). The proposed mechanism to guarantee software confidentiality is built on top of a Ciphertext-Policy Attribute-Based Encryption (CP-ABE) scheme [18].

A preliminary version of this paper appeared as [1], where the key material is associated to each *package*. In this extension we associated keys to *features*. This choice allows a more flexible definition of dependencies.

2.1 Solutions and issues using current ABE schemes

Attribute Based Encryption (ABE) schemes [19] allow a user to decrypt a given ciphertext only if a set of attributes matches the ones described by the decryption policy.

ABE comes in two flavours. In Key-Policy ABE (KP-ABE) the decryption is possible if the attributes encoded in the cipher text satisfy the decryption policy specified by the secret key of the user. In Ciphertext-Policy-ABE, or CP-ABE, the decryption is possible if the attributes encoded by the secret key of the user match the decryption policy specified by the cipher text.

In this context, as it has been done in [17], a possible solution is to use CP-ABE scheme to encrypt the software package before its distribution. In this way the vendor can define the software decryption policy in the moment in which the package is published. At the same time there is no need to provide users with new key material. Indeed a user will be able to decrypt the package if she holds a set of attributes that meets the policy specified by the vendor.

This solution has a number of pros. First of all, starting from [20], there exist CP-ABE scheme in which the vendor can specify arbitrary (i.e., possibly non-monotonic) release policies. Notice that, in the context of software distribution, non-monotonic access structures can be used to describe *conflicts* between packages, i.e., the requirement that one package can be installed only if another one is *not installed*. Unfortunately, non-monotonic CP-ABE schemes require that the user receives a key element for every possible attribute. Since, in our interpretation, attributes correspond to software packages, this means that each user/device should receive a key element for *each possible package*.

Furthermore, the existence of Distributed CP-ABE schemes, first introduced in [21], allow software vendors to distributively compute the secret keys representing the attribute each user holds.

Thus, in a *static* setting in which the user attributes do not frequently change over time and in a setting in which either there exists a single software vendor or a few ones that are willing to cooperate, the usage of CP-ABE can be used to solve efficiently the problem of secure software distribution.

In this paper we consider a more *dynamic scenario*. Each time an administrator/user installs a new package on a device, such a software enriches the device with a new set of *attributes*. This is immediate, for example, in the case in which the installed package is a library. Indeed, the presence of the new library makes possible the installation of all the software packages that depend on it and whose installation was impossible before. In this respect, the installation of *each new software* modifies the set of attributes of a given system. If we look at the key management problem from this point of view, we can claim that it cannot be considered at all a static process.

It is thus crucial to provide efficient procedures for the update of the key material that users hold in response of a software installation. Unfortunately, in existing CP-ABE schemes either there must exists a single trusted centralised authority that generates the keys for each user or there are multiple, possibly independent, authorities [22] but each user has a unique *Global Identifier (GID)*. In the latter case, the user key generation requires the user to provide the GID and all his attributes to the authority. This, in turn, would allow a complete profiling of the user device that is, in principle, a serious privacy issue and constitutes by itself a security threat. In PPDCP-ABE [23], the authors try to mitigate information leakage. Unfortunately, in this scheme the secret key generation algorithm requires interaction between the authority and the user.

3 Package Management

At glance, software deployment is the complex of activities pursuing components retrieval, installation and configuration on a computer system. Such components include software with many different purposes such as implementing system-wide services, features and tools (such as kernel modules, device drivers, dynamic libraries); new user applications rather than *updates* and *upgrades* for components already present on the system. Nowadays, major operating systems, as well as several frameworks and application suites, are conceived to operate within a *Package Management System (PMS)* , a distributed infrastructure which carries out the activities related to the deployment of software components. Main PMS activities are briefly described in the following paragraphs.

Once its development process is completed, a new version of a certain component is made available. The developer/vendor (*issuer*) releases a deployment unit (*package*) that consists of the component itself, along with the information to verify and install it (*metadata*). Packages can be published on the issuer's repositories as well as disseminated through a set of third-party *mirrors*, each of which replicates the content of the official repository.

Metadata have the twofold purpose of: (a) unambiguously identifying the component, its version, etc. and (b) declaring a set of properties and requirements (including dependencies) that have to be satisfied in order to correctly install the component. Metadata can be either shipped along with the software they refer to, or distributed separately. An official "components directory" is generally provided to publish at least metadata of the released package along with the link to the repository from which it can be downloaded.

The owner/administrator of the target system (*consumer*), is enabled to search for the requested component by querying the directory. Query fields include package's name, version, target system etc. Once she has obtained the information about the package, the consumer may choose whether or not to install the suggested components, according a *local* policy which takes into account several package's properties such as release date, freshness, provenance, license terms etc. Usually, the consumer is provided with a dedicated set of utilities that implement all deployment tasks. In the following such utilities are referred to as *package manager*.

The installation process concerns adding any new components to the target system that provide new features, whereas updates and upgrades that are about changing the setup of already installed components. Once a package is downloaded, the package manager verifies its integrity in order to identify any transmission error before the installation process begins. Furthermore, the authenticity of the package should be checked, in order to prevent the installation of counterfeit code.

Then, the package manager evaluates the package's *pre-installation* requirements, that mainly concern of target device properties (*e.g.*, OS type/version, processor) and dependencies to already installed packages. Package management systems feature a local *package inventory* that stores the metadata of every installed package. The package manager uses it to check the satisfiability of dependencies. Eventually, metadata of the incoming package are added to the inventory. If all requirements are satisfied, the package manager copies all files contained in the package to their destination.

3.1 Security in package management systems

Following [5], in order to improve their security, software distribution systems started to redefine the distribution protocols. In particular, the following guidelines are implemented by secure distribution services:

Package Authentication. Guarantee the (actual) usage of digital signatures.

Multiple protection levers. Provide, if possible, multiple levels of protection. Most repositories also provide a special *root metadata* file that stores the position in the file system and a secure hash for each package in the repository. The signature of root metadata coupled with the package signature or (package) metadata signature can provide different levels of security and usability.

Freshness Verification. Verify the *freshness* of metadata files. Outdated metadata files, even if such files are signed, might induce the user to install insecure software packages.

Multiple sources. Execute software updates from multiple sources, e.g., by checking the metadata on one repository and downloading the software package from another one. This procedure might allow the identification of malicious repositories.

4 System Model and Requirements

We consider a model in which we distinguish three different players. Our model resembles the current Linux software distribution system. We will also describe the security requirements guaranteed by our system.

4.1 System Model

Software distributions/updates are carried out by means of a Distribution Server (DS) and possibly a set of Mirror Server (MS). The role of the Distribution Server is to generate properly formatted and encrypted software packages, containing the actual software to be installed, the installation policy, i.e., the specification of the set of required packages, along with the software metadata. Furthermore, the DS has to manage the timely and efficient distribution of the encrypted packages to the set of Mirrors. The Distribution Server (DS) is managed by the software vendor and it will be considered trusted. The role of the Mirror Servers is to answer requests sent by users and provide the required packages. The Mirror Servers (MS) are typically managed by third parties and, for this reason, MSes will be considered untrusted. Finally, the Users can query either one MS or the DS directly. As suggested in [5], and as it is currently done by some package distribution systems, the user might download for security reasons the metadata file from the DS while, for performance issues, the download of the software package is done via one MS.

The target device features a *legacy operating system* (LOS) and a *package management subsystem* (PMS) that provides all functionalities related to the decryption and the evaluation of the installation policy and maintenance of the database of *feature keys*. The LOS features both normal and privileged users, whereas the *PMS administrator* is in charge of the PMS configuration and management. LOS and PMS are assumed to be mutually separated and communicate by means of a set of APIs. LOS and PMS administrators are two distinct roles. This should be not regarded as an unnatural assumption. For example, Windows Operating Systems, feature two privileged accounts: "administrator" and "SYSTEM". The former is intended to be used by the system manager for performing every operation except those affecting system's integrity *e.g.*, programs setup and removal. The latter can only be used non-interactively for package management and through a strictly manual procedure.

Package processing outputs *contents* (*i.e.*, executables, configuration files, libraries etc. composing the installed software) and *metadata* (*i.e.*, the information related to package version, author, release notes, verification tokens, etc.) along with some "private" metadata, concerning key material used to decrypt installation policies etc. (feature keys). Package contents are stored on the LOS filesystem, package metadata are stored in the PMS.

In facts, the PMS relies on a private secure storage where feature keys (along with any package-related private or verification information) are stored and never made available to any other system component (including LOS administrators).

Currently, plenty of technologies are available to implement such PMS feature. The secure storage can be built on top of cryptographic tamper-resistant devices such as SIM cards or other so-called Secure Elements at large, either embedded or plugged on-demand into the platform. On the other hand, target systems are required to ensure the trustworthiness of the PMS operation. Again, to this end, plenty of solutions are currently on the shelf, e.g., leveraging on Trusted Computing-powered operating systems or the forthcoming Trusted Execution Environment architecture where the PMS along with its key store naturally fits as "Trusted Application" [24].

4.2 Security Requirements

We define in this section the security requirements the proposed system guarantees.

Confidentiality. One of the key issues in software distribution is the confidentiality of the software. We will focus on the software protection from the moment in which the package is created by the issuer to the moment in which it is decrypted on the target device. Indeed, after its decryption, the software might be maliciously distributed at will.

Authenticity and Integrity. Another fundamental problem in software distribution/update is guaranteeing the authenticity and integrity of the installed software. The proposed solution will allow each user to *locally* verify that the software that she has downloaded meets such properties.

Freshness. As stated in the previous section, one of the key requirement for secure software update is guaranteeing the freshness of the packages. As an example consider the update of the database containing virus definitions. In this case, the freshness of the software component is crucial for guaranteeing the system protection. Typically a new *fresh* package outdates all its previous versions/releases. We do not provide a formal definition of freshness as our system allows its formulation as an arbitrary combination software issuer and user defined policies. Finally, each issuer has the possibility/duty to decide its own policy for defining freshness for its own products.

Strong dependencies enforcement. We require that an adversary is not able to force the installation of a software on a system that does not meet all the prerequisites. The enforcement is *strong* in the sense that, if some required dependency has not been installed, the package cannot even be decrypted.

4.3 Adversary Model

In this paper we consider the case in which the Distribution Server (DS) is trusted by the users. Notice that, although we consider the case in which multiple independent distribution servers exist, from the point of view of the adversary, the existence of one or multiple of such servers is immaterial. Indeed each DS is assumed to work independently from the others. The model also assumes the existence of Mirror servers whose only role is to store encrypted packages generated by the DS and forward them to the users whenever they require it. The Mirror Servers are untrusted, as they might try to gain confidential information from the stored packages or send corrupted packages to the users. As in [17], the adversary has complete access to the communication network, i.e., she can manipulate, create or duplicate legitimate messages.

An adversary has LOS administrator privilege (including full access file systems and networking facilities). In particular the adversary can: (a) access and modify every file included in the package content (*i.e.,* any file or component once it has been installed); (b) "manually" install (import) a package by copying its content as it is installed on a different device and (c) control/deviate the connection with the package repositories, in order to force the PMS to install malicious packages carrying fake features.

However, the adversary cannot alter the behaviour of the package management subsystem itself. That is, she can neither read, insert or delete feature keys nor modify vendors' public keys nor modify any other information stored by the package management subsystem. To this aim, the system uses a tamper-proof secure storage area, which we refer to as the *package inventory* that is readable and modifiable only by the PMS.

The main goal of the adversary is to maintain/translate the target device in/to a state in which certain "malicious" activities are possible or certain features are illegitimately available. More precisely, the adversary aims at making the system either to install a package despite the system does not satisfy all its dependencies or to deny the installation of a package by "simulating" a dependency failure or forge the image of the device setup, in order to pursue any vendor support contract and license infringements.

4.4 A note on the application scenario

As stated in the introduction, the purpose of our system is twofold. On one hand, it guarantees the authenticity and integrity of the package that is installed on a secure system. On the other hand it forbids, even to the administrator, the installation of a package for which dependencies are not met. The latter requirement can be intended in one of the following ways: The package to be installed depends on a package that (a) has never been installed on the system. or (b) has been installed on the system but the adversary has modified it, *after its installation* e.g., by deleting files. We completely cover case (a) as our system prevents the installation.

We point out that verifying the integrity of a package *after* its installation (*e.g.* at run-time) is not up to the PMS, as more effective solutions are available to this end [25]. In particular, we mention IMA [26], which is one of the earliest and most influential proposals in this field, along with its proposed improvements. IMA extends the *chain of trust* beyond the components involved in the bootstrap process, to the whole platform's run-time.

Our PMS strengthens dependency enforcement and provides a mechanism to ensure package confidentiality in case its dependencies are not satisfied. Moreover the PMS, *before* installing a new package, verifies the integrity of those already installed features it depends upon. Having done so, its job is over. Nevertheless, our system does not requires to be used necessarily in conjunction with any integrity verification tool, though it could give them an added value, as it could provide a further criterion to contain their verification scope.

In our system, features and dependencies form a hierarchy that roots in a set of features that do not explicitly depend on any other features. However every single feature implicitly depends at least on the base system, which includes the PMS itself. In our infrastructure, we made explicit this dependency in every package, so that, once the base system installation is completed, the PMS is already configuread and its features keys repository contains at least the base system's key (feature key zero). Therefore, packages which *apparently* have no requirements are actually encrypted with their package key which is, in turn, encrypted with such a key.

The feature key zero is stored on a secure element (*e.g.*, a smart card), it is never released to the administrator and is provided only to the Operating System installer at setup time, through a secure protocol. To this end, we rely on the base assumption that the target platform provides secure boot facilities (*e.g.*, UEFI's secure boot [27] and Global Platform's TEE secure boot [28]).

5 Preliminaries and Notations

Secret sharing schemes are one of the building blocks we use in our solution. Let P be a set of players and let $D \notin P$ be a special player called the dealer. An access structure $\mathcal{A}$ over P is a monotone collection $\mathcal{A} \subseteq 2^P$. The basis of an access structure $\mathcal{A}$, denoted by $\delta(\mathcal{A})$, consists of the collection of its *minimal sets*, i.e., $\delta(\mathcal{A}) = \{A \in \mathcal{A} \mid \forall A' \subset A, A' \notin \mathcal{A}\}$. In our description, each software package corresponds to a *player* in the secret sharing terminology while an installation policy corresponds to an access structure.

It is well known that secret sharing schemes exist only for monotonic access structures. There exists a natural correspondence between the family of access structures and monotone boolean formulae [29]. We will denote by $\mathcal{A}_\phi$ the access structure, defined over a set P of n players, representing the monotone boolean formula ϕ defined over a set of n variables. Specifically, a set $A \subseteq P$ is authorised in $\mathcal{A}_\phi$ if and only if it corresponds to a truth assignment for ϕ. We note that, for every access structure $\mathcal{A}$, its basis $\delta(\mathcal{A})$ can be trivially mapped to a DNF formula by associating each authorised set in $\delta(\mathcal{A})$ to a clause in the formula ϕ.

Formally, a secret sharing scheme [30] consists of a pair of efficient algorithms, a distribution algorithm $\texttt{Distribute}(\cdot,\cdot)$ and a reconstruction algorithm $\texttt{Reconstruct}(\cdot,\cdot)$. The $\texttt{Distribute}(\mathcal{A}, s)$ algorithm takes as inputs an access structure $\mathcal{A}$ over a set P and a secret s and outputs a set of shares $S = (s_1, \ldots, s_n)$, one for each player in P. The reconstruction algorithm $\texttt{Reconstruct}(\mathcal{A}, \hat{S})$ takes as inputs the access structure $\mathcal{A}$ and the set $\hat{S} \subset S$ of shares held by the players in $\hat{P} \subset P$. If $\hat{P} \in \mathcal{A}$ than the reconstruction algorithm outputs the secret s otherwise it outputs a special symbol $\perp$. Secret sharing schemes that work for *every* access structure exist, e.g., [31], but they may result in inefficient distributions. At the same time efficient secret sharing schemes have been developed for specific classes of access structure, e.g., [33]–[35].

We assume that each software issuer has a key pair (Sk_V, Pk_V) that are used to generate digital signatures. Given a message m and a signature $\sigma = \texttt{Sign}_{Sk_V}(m)$, we denote by $\texttt{Verify}_{Pk_V}(m, \sigma)$ the verification algorithm that either accepts the signature or outputs a special symbol $\perp$. The protocol uses a symmetric encryption scheme. We will denote by $\texttt{Encrypt}_k(\cdot)$, (resp., $\texttt{Decrypt}_k(\cdot)$) the encryption (resp., decryption) function under the secret key k. We assume that the encryption scheme used in the protocol meets the following properties: $\forall m$ and $\forall k$, $\texttt{Decrypt}_k(\texttt{Encrypt}_k(m)) = m$ and, $\forall m, \forall k, k', k \neq k'$ it holds that $\texttt{Decrypt}_k(\texttt{Encrypt}_{k'}(m)) = \perp$. The functions $\texttt{CreateMetaData}()$ and $\texttt{ExtractMetadata}()$ are used to create and extract the metadata information for a given software bundle, respectively.

6 The protocol

In this section we describe the proposed protocols. Our system consists of two procedures. The first one, executed by the Distribution Server, is used by the software vendor to create an encrypted and signed package E and its associated metadata M_E. Such package will be then distributed to the Mirror Servers. A second procedure, executed by the user, takes as input the signed package E and its metadata and either installs the software therein contained or rejects it.

6.1 The protocol: An informal description

In our proposal, the set of ciphertexts that has to be decrypted consists of the set of encrypted software packages. A system should be able to decrypt a given package only if it holds all the necessary *attributes* that are needed to execute the software therein contained. Furthermore, the package itself should contain some information that allows to update the set of attributes. Such a modification should be done *non-interactively*.

Whenever a new software package is installed on a given system, we can consider the new capability provided by the software as a new *attribute* of the system. Following the idea underlying ABE schemes, this new attribute should be used to update the set of keys of the system in order to extend the set of ciphertexts that can be decrypted by the system.

In our solution each package p is identified by a unique name n. The package p is assumed to *require a set of features* $\mathcal{R} = \{f_1, \ldots, f_m\}$ and to *provide* a set of features, $\mathcal{P} = \{\hat{f}_1, \ldots, \hat{f}_{\hat{m}}\}$. Each feature is associated to a random key which we call the *feature key*. The set of keys provided by the packages installed on the current system are stored in a *package inventory*, which we denote by $\mathcal{I}$. As stated in Section 4.3, the information in $\mathcal{I}$ cannot be read or altered by the adversary. Every package p that depends on the feature f_i should be encrypted with a random *encryption key* r that can be reconstructed if and only if the feature key k_i, associated to f_i, is known.

Informally, the distribution protocol works as follows. Consider a package p that requires *all* the features $f_1, \ldots, f_m$. In this case the installation policy can be described by an access structure called m-out-of-m threshold access structure and denoted by $\mathcal{T}(m, m)$. Let us denote by k_i the feature key associated to feature f_i. The vendor generates a random encryption key $r = r_1 + \ldots + r_m$ and computes $e_i = \texttt{Encrypt}_{k_i}(r_i)$. In other words she encrypts the random subkey r_i of the *encryption key* r for package p with the *feature key* k_i of feature f_i. The vendor builds a software package p by putting together the software, all the necessary metadata and all features keys associated to the features $\hat{f}_1, \ldots, \hat{f}_{\hat{m}}$ provided by the package p. This package is encrypted with the encryption key r, i.e., $E_r(p) = \texttt{Encrypt}_r(p)$. The vendor publishes the encrypted package along with $e_1, \ldots, e_m$, i.e., $E = ((f_1, e_1), \ldots, (f_m, e_m), \mathcal{T}(m, m), E_r(p))$.

When the package p needs to be installed, the user downloads E. In this case, if the user holds all the feature keys $k_1, \ldots, k_m$, associated to all the required features, she can (a) decrypt every e_i, (b) reconstruct the key r and (c) properly decrypt $E_r(p)$. If the system is missing any of the prerequisite, she will not be able to execute the decryption and, thus she will not be able to install the package. Once the package has been decrypted, the device holds the keys associated to the features provided by the installed package and, from this point on, it is possible to install the software packages that depend on them.

6.2 A protocol for monotone formulae

The protocol described in the previous section solves the problem of enforcing prerequisite matching in software installation whenever the installation policy requires *all the features in the set* $\mathcal{F} = \{f_1, \ldots, f_m\}$, i.e., the installation policy can be described by a single-clause DNF formula $\phi = (f_1 \wedge \ldots \wedge f_m)$.

Clearly this type of formula is not expressive enough to describe arbitrary installation policies. As an example, it cannot express an installation policy that requires the presence of feature f_i in one to its allowed versions $v_1, \ldots, v_m$, i.e., $f_{i,v_1} \vee \ldots \vee f_{i,v_m}$. Notice that having the possibility of expressing this type of installation policy is crucial as this is a typical way to express backward compatibility w.r.t. *some* previous versions of a given functionality.

Let p be a software package and let ϕ its installation policy. In the following we will assume that ϕ is described by a *monotone* boolean formula. The formula can be seen as an access structure $\mathcal{A}_\phi$ defined over the set of required features $\mathcal{R} = \{f_1, \ldots, f_m\}$. Let us denote by $\{k_1, \ldots, k_m\}$ the feature keys associated to the required features in $\mathcal{R}$. Furthermore, let $\mathcal{P} = \{\hat{f}_1, \ldots, \hat{f}_{\hat{m}}\}$ be the set of features provided by the package p and let $\{\hat{k}_1, \ldots, \hat{k}_{\hat{m}}\}$ the corresponding feature keys.

Notice that, from the practical point of view, if a feature $f \in \mathcal{R}$ is required for the installation of package p, it has to be the case that the vendor has already installed some package that provides the functionality f. This assumption is quite natural in contexts in which the software to be released has to be quality certified. For example, let us consider the Common Criteria Recognition Agreement [36], a *de-facto* standard for the certification of security-related products. CCRA software quality testing is executed on a specific software *configuration* or *set of configurations*. Roughly speaking, a configuration identifies, among others things, all software components, including the specific version of each external package the software uses, along with their implementation. We can thus safely assume that the software vendor knows the feature key k_f. Would it be otherwise, she could not have even created and tested her own software.

We further observe that for the functionalities $\hat{f}_i \in \mathcal{P}$, two cases may arise. Either a feature $\hat{f}_i$ is a new functionality, implemented by p for the first time, or it already existed in some other package, e.g., in a previous version of the same package p. In the former case, the corresponding feature key $\hat{k}_i$ is randomly generated at the package creation time. In the latter case, as for the keys associated to the required features, we can assume that they are known to the software vendor. In any case, we can assume that the software vendor is able to obtain all the feature keys associated to the features in $\mathcal{R}$ and $\mathcal{P}$.

At package creation time, the set of keys associated to the features provided by the current software has to be embedded in the encrypted package. The software issuer extracts the feature keys associated to existing features from the package inventory. At the same time, she generates random feature keys for new ones and updates the inventory accordingly. Algorithm 1 describes the construction of such set of keys.

The package creation process proceeds as follows. The software vendor generates a random encryption key r and shares it among the required features in $\mathcal{R}$ by using an efficient secret sharing scheme for $\mathcal{A}_\phi$. In this way, each feature $f_i \in \mathcal{R}$ will have an associated share s_i. Each such share is then encrypted with the feature key k_i, i.e., $e_i = \texttt{Encrypt}_{k_i}(s_i)$. The sequence $R = (f_1, e_1), \ldots, (f_m, e_m)$ will be included in the package. Intuitively, each device holding the feature keys associated to all the features required by authorized set in the installation policy will be able to recover r, otherwise she will obtain no information about it.

Algorithm 1 Generation of the set of Provided Features Keys

```
function GenerateProvidedKeys($\mathcal{I}, \mathcal{P}$)
    $\mathcal{K} \leftarrow \emptyset$
    $\hat{m} \leftarrow |\mathcal{P}|$
    for $j \leftarrow 1$ to $\hat{m}$ do
        $\hat{k}_j \leftarrow \texttt{Extract\_Key}(\mathcal{I}, \hat{f}_j)$
        if $\hat{k}_j = \perp$ then
            Generate a random feature key $\hat{k}_j$
            $\mathcal{I} \leftarrow \mathcal{I} \cup (\hat{f}_j, \hat{k}_j)$
        end if
        $\mathcal{K} \leftarrow \mathcal{K} \cup (\hat{f}_j, \hat{k}_j)$
    end for
    return $(\mathcal{K}, \mathcal{I})$
end function
```

The vendor then generates a timestamp t and creates a package containing the software s, the software name n, the set of provided features keys pairs $\{(\hat{f}_1, \hat{k}_1), \ldots, (\hat{f}_{\hat{m}}, \hat{k}_{\hat{m}})\}$, the timestamp t, a freshness interval length Δ, the software metadata M_s. This package is encrypted by using the encryption key r. The encrypted package, along with the sequence R of encrypted shares and the installation policy $\mathcal{A}_\phi$ are packed together to obtain the E.

The vendor creates metadata information for E by including the package name n, the timestamp t, the software metadata M_s and other required information for E, e.g., size, hash value, creation date, extra information, etc. Notice that once the package is encrypted, the package name, the timestamp and metadata M_s are bound to the software but, at the same time, they are not accessible without decrypting it. Since such information is also needed in the clear in order to speedup integrity checks, we add them both the E and M_E. Finally, the vendor signs separately E and M_E and send to the Mirrors. Algorithm 2 describes the protocol executed by the vendor.

Algorithm 2 Distribution Server

```
procedure CreatePackage($Sk_V, s, M_s, \mathcal{I}, \mathcal{R}, \mathcal{P}, \mathcal{A}_\phi, \Delta$)
    Generate a random encryption key $r$                ▷ Key Sharing
    $(s_1 \ldots, s_m) \leftarrow \texttt{Distribute}(r, \mathcal{A}_\phi)$
    Parse $\mathcal{R}$ as $\{f_1, \ldots, f_m\}$
    for $j = 1$ to $m$ do                               ▷ Shares Encryption
        $k_j \leftarrow \texttt{Extract\_Key}(\mathcal{I}, f_j)$
        $e_j \leftarrow \texttt{Encrypt}_{k_j}(s_j)$
    end for
    $t \leftarrow \texttt{TimeStamp}()$        ▷ Encrypted Package Generation
    $(\mathcal{K}, \mathcal{I}) \leftarrow$ GenerateProvidedKeys$(\mathcal{P}, \mathcal{I})$
    $p \leftarrow (n, t, \Delta, M_s, \mathcal{K}, s)$
    $E \leftarrow \langle\langle (f_1, e_1), \ldots, (f_m, e_m)\rangle, \mathcal{A}_\phi, \texttt{Encrypt}_r(p)\rangle$
    $M_E \leftarrow \texttt{CreateMetaData}(n, t, \Delta, M_s, E)$
    $\sigma_M \leftarrow \texttt{Sign}_{Sk_V}(M_E)$
    $\sigma \leftarrow \texttt{Sign}_{Sk_V}(E)$
    Send $(E, \sigma), (M_E, \sigma_M)$ to Mirror Servers
end procedure
```

The key reconstruction algorithm takes as inputs the bundle E and the package inventory $\mathcal{I}$. The user extracts from E the sequence $(f_1, e_1), \ldots, (f_m, e_m)$ and the installation policy $\mathcal{A}_\phi$. She decrypts all the shares she can by using the feature keys contained in the local key inventory $\mathcal{I}$ and, if possible, she reconstructs the encryption key r.

The installation procedure described by Algorithm 3 is executed by the user. It takes as inputs the metadata (M_E, σ_M), signed by the issuer and downloaded either from the Distribution Server or from one Mirror Server; the signed package (E, σ_E); the issuer public key and the local package inventory. As a preliminary step, the user checks the signatures on these packages by using the verification key of the software vendor. The user checks that the bundle E matches the metadata information contained in M_E. Furthermore, as anticipated in Section 4.2, depending on the *local* and on the *issuer* policies, she checks the freshness of the package. If any of the above checks fails, the installation is rejected. The key reconstruction algorithm is used to recover the decryption key and the software package is decrypted. If the decryption is successful, the user extracts the software metadata M_s from M_E and verifies their correspondence. In this phase, the procedure also checks, e.g., by using the metadata stored by the PMS, if the implementations of the features required by the package have not been altered. If all tests pass, she installs the software and adds the features keys contained in E to the package inventory.

Algorithm 3 User Installation Procedure

```
procedure INSTALL($(M_E, \sigma_M), (E, \sigma), Pk_V, \mathcal{I}$)
    if ($\texttt{Verify}(E, \sigma) = \bot$)$\vee$($\texttt{Verify}(M_E, \sigma_M) = \bot$) then
        Reject                                   ▷ Authenticity Verification
    end if
    if ($E$ does not Match $M_E$) $\vee$ ($M_E$ Not Fresh) then
        Reject                      ▷ Integrity and Freshness Verification
    end if
    Parse $E$ as $\langle((f_1, e_1), \ldots, (f_m, e_m)), \mathcal{A}_\phi, S\rangle$
    $r \leftarrow \texttt{ReconstructKey}((f_1, e_1), \ldots, (f_m, e_m)), \mathcal{A}_\phi, \mathcal{I})$
    $p \leftarrow \texttt{Decrypt}_r(S)$                         ▷ Package Decryption
    $M_s \leftarrow \texttt{ExtractMetadata}(M_E)$
    if ($p \neq \bot$) $\wedge$ ($p$ matches $M_s$) then
        Parse $p$ as $(n, t, \Delta, M_s, \mathcal{K}, s)$
        Check required features integrity on disk
        Install $s$                                ▷ Software Installation
        $\mathcal{I}' \leftarrow \mathcal{I} \cup \mathcal{K}$                  ▷ Package inventory update
    end if
end procedure
```

6.3 Security Analysis

In this section we provide a security analysis w.r.t. the security requirements presented in Section 4.2.

Confidentiality. The confidentiality of the software in the encrypted package is guaranteed by means of a symmetric encryption scheme. Since we assume that such a scheme is secure, the confidentiality of package relies on the impossibility of extracting the encryption key r from the encrypted shares $\langle(f_1, e_1), \ldots, (f_m, e_m)\rangle$. Here, again, the security of each share is guaranteed by the security of the symmetric encryption scheme used to encrypt it and by the security of the package inventory. Furthermore, for every set $A \subseteq N$ such that $A \notin \mathcal{A}_\phi$, since secret sharing scheme is perfect, the user can obtain no information on the decryption key given all the decrypted shares associated to the packages in A. This means that the user can obtain the encryption key if and only if she holds all the feature keys associated to required packages.

Authenticity and Integrity. Software integrity is guaranteed by the use of secure hash functions in the generation of the metadata associated to the software and for the one associated to the encrypted package E. Secure signature schemes are used to guarantee the authenticity of the information that the Distribution Server sends to the Mirror Servers and that these ones forward to the Users.

Freshness. The encrypted package E and its associated metadata M_E include a timestamp t, generated by the Distribution Server, and the length of the validity period Δ. From the point of view of the issuer the package has to be intended to be fresh in the interval $[t, t + \Delta]$. On the other hand the freshness of a package can be influenced by the installation policy of the user. For example, a PMS administrator may use the following policy: *Install security packages immediately but install other packages only when the version is stable.* For this reason we do not bind the freshness to the issuer-generated freshness interval but we allow the user to override this interval whenever its local policy requires to.

Strong dependencies enforcement. The user installation procedure guarantees the dependency enforcement under the assumption that the local package inventory has not been modified. Indeed, under such assumption, the user will not be able to decrypt the downloaded package unless she meets the issuer-defined installation policy. Furthermore, the procedure also verifies that feature implementations have *not* been altered *after* their proper installation. We stress that if we start from a safe system, the strong dependency enforcement guarantees that the installation does not bring the device in an unsafe state that, in turn, guarantees the inventory integrity. Furthermore, we can relax this assumption by requiring the administrator to sign the local inventory after each installation.

7 Two policy-hiding protocol extensions

The protocols presented in the previous section assume that the installation policy is public. However, there might be cases in which the installation policy, or even part of it, should not be public. Consider, for example, the case in which the software package has to be installed on a flock of military drones. The installation policy might express, among other things, requirements regarding the set of devices that are required to complete the mission. In this case, the installation policy itself might release sensitive information, e.g., it is clear that the goal of a mission requiring a *weapon* device is different from the one requiring a *camera* device. In this context, our extensions hide (part of) the installation policy to every user who *does not* have the right to install the package.

In order to meet this additional security requirement, in this section we present two policy-hiding protocol extensions. The two extensions aim at hiding different information regarding the installation policy. More precisely, in the first extension the structure of the installation policy is public. On the other hand, either the adversary holds the feature key needed to decrypt a given share s_i generated during the package creation process or she cannot tell to which feature f_i that share is associated to. In the second extension, the formula structure itself will be hidden from the adversary.

7.1 Policy hiding: An overview

In order to provide policy-hiding protocols, we need to take special care of two different issues. First of all, the protocols should not release feature names in the clear in the package. Second, and more important, is the fact that secret sharing schemes assume the access structure to be public. Secret sharing schemes exist for *general*, i.e., arbitrary monotone, access structures and for *specific*, e.g., threshold, or graph-based, access structures. In the former case the reconstruction algorithm requires as input the access structure to properly combine the shares. General schemes sacrifice efficiency (in terms of share size) for generality. In the latter case, the reconstruction algorithm might not require the access structure as input. On the other hand, since schemes are designed for specific classes of access structures, they are more efficient than general ones at the cost of loosing generality and releasing the access structure topology.

In both cases, the protection of the installation policy is sacrificed in favour of efficiency considerations. Our idea underlying the protection of the policy is to use a less efficient solution in favour of a more secure one.

7.2 An efficient protocol for partially-hidden policies

A first step toward policy protection is the *anonymization* of feature names. The package creation procedure includes feature names in the installation policy specification. The key idea is to remove feature names from the package. Notice that, such a protocol modification does not hide the installation policy but clearly hides the correspondence between an encrypted share and its required feature both to a legitimate user and to an adversary. Since references to feature names are missing, the user cannot recover the proper key by looking up in package inventory but she needs to scan such database and try to decrypt the current share with each key therein contained. At first sight, this might turn out to be quite inefficient as the decryption of each share takes a time that grows with the number of feature keys stored in the inventory. However, as we will see in the next section, in practice the number of features included in a typical system configuration rarely exceeds one thousand, so that the scanning time remain quite reasonable (about few seconds). Such decryption strategy works if the user is able to distinguish between the case in which she is using the correct key from the case in which the key she is using for the decryption is wrong.

Clearly, an adversary that receives a package, can recognise the structure of the installation policy and she can decrypt each share for which she knows the corresponding key. On the other hand, for each share for which she does *not* know the corresponding feature key, the adversary cannot say which is the required feature she is missing.

7.3 A protocol for fully-hidden policies

In order to further reduce the amount of information leaked by package, we present a protocol that also hides the structure of the installation policy. More specifically, we will use a *standardised* access structure defined over a set of *anonimized* feature names. Intuitively, standardised access structures cannot be used for secret sharing scheme designed for specific classes of access structures. At the same time, as before, anonymised feature names do require a linear search in the package inventory for retrieving the proper feature key, in place of the direct access given the feature name.

Let ϕ be the monotone formula defined over the variables associated to the required features $\mathcal{R} = \{f_1, \ldots, f_m\}$. We first anonymise the set of names, by substituting each name in $\mathcal{R}$ with its corresponding index i, i.e., $\hat{\mathcal{R}} = \{1, \ldots, m\}$. We stress that this mapping is performed locally (and arbitrarily) by the dealer/software vendor. We then convert the formula ϕ into its Disjunctive Normal Form. In this way we transform a monotone boolean formula ϕ defined over $\mathcal{R}$ into a monotone boolean formula in DNF $\hat{\phi}$ defined over $\hat{\mathcal{R}}$. Given a DNF formula $\hat{\phi} = C_1 \vee \ldots \vee C_k$, where $C_i = x_{j_1} \wedge \ldots \wedge x_{j_i}$, it is possible [31], [33] to additively share the encryption key r, independently for each C_i, and obtain shares $s_{i,j_1}, \ldots, s_{i,j_i}$. At this point, as in the previous protocol, each share s_{i,j_h} associated to feature j_h is encrypted with the feature key k_{j_h}, i.e., $E_i = (\texttt{Encrypt}_{k_{j_1}}(s_{i,j_1}), \ldots, \texttt{Encrypt}_{k_{j_i}}(s_{i,j_i}))$. The Distribution Server publishes $E = ((E_1, \ldots, E_k), \texttt{Encrypt}_r(p))$. Algorithm 4 describes the whole protocol executed by the Distribution Server. Notice that E does *not* contain a description of the installation policy in clear. The information that can be inferred by this representation is the number of clauses in the installation policy and the number of literals in each clause.

Algorithm 4 Anonimized Distribution Server

```
procedure AnonPkgCreation($s, M_s, \mathcal{I}, \mathcal{R}, \mathcal{P}, \Delta, \mathcal{A}_\phi$)
    Generate a random encryption key $r$
    Parse $\mathcal{R}$ as $\{f_1, \ldots, f_m\}$
    for all $A_i = \{j_1, \ldots, j_{l(i)}\} \in \delta(\mathcal{A}_{\hat{\phi}})$ do
        Randomly select $(s_{i,j_1} \ldots, s_{i,j_{l(i)-1}})$
        $s_{i,j_{l(i)}} \leftarrow r \oplus \bigoplus_{w=1}^{l(i)-1} s_{i,j_w}$
        for $w \leftarrow 1$ to $l(i)$ do
            $k_{j_w} \leftarrow \texttt{Extract\_Key}(\mathcal{I}, f_{j_w})$
            $e_{j_w} \leftarrow Encrypt_{k_{j_w}}(s_{i,j_w})$
        end for
        $C_i \leftarrow \langle e_{j_1}, \ldots, e_{j_{l(i)}} \rangle$
    end for                              ▷ Encrypted Package Generation
    $t \leftarrow \texttt{TimeStamp}()$
    $(\mathcal{K}, \mathcal{I}) \leftarrow$ GenerateProvidedKeys$(\mathcal{P}, \mathcal{I})$
    $p = (n, t, \Delta, M_s, \mathcal{K}, s)$
    $E \leftarrow \langle \langle C_1, \ldots, C_{|\mathcal{A}|} \rangle, Encrypt_r(p) \rangle$
    $M_E \leftarrow \texttt{CreateMetaData}(n, t, \Delta, M_s, E)$
    $\sigma_M \leftarrow \texttt{Sign}_{Sk_V}(M_E)$
    $\sigma \leftarrow \texttt{Sign}_{Sk_V}(E)$
    Send $(E, \sigma), (M_E, \sigma_M)$ to Mirror Servers
end procedure
```

The above transformation forces the modification of the Key Reconstruction Phase in the installation procedure, too.

Indeed, when the user downloads the software package, because of the anonymization of feature names, she does not know which installed key should be used to decrypt a given share. Thus the procedure has to decrypt each share with each installed key. Clearly since shares are random, we need to make again the loose assumption that the user can distinguish the decryption executed with the wrong key from a decryption executed with the correct key. Upon retrieving a software package, for each encrypted clause E_i the user tries to decrypt each encrypted share $\texttt{Encrypt}_{k_{j_1}}(s_{i,j_l})$ using all the keys she has in her local package inventory. If she fails in decrypting one share, she moves to the next clause. If she manages to decrypt all the shares in a given clause, she can reconstruct the encryption key for the package and install the software.

Algorithm 5 Decryption Key Reconstruction Phase

```
function AnonymizedReconstruction($(E, \sigma), \mathcal{I}$)
    Parse $E$ as $\langle\langle C_1, \ldots C_{|\mathcal{A}|}\rangle, \texttt{Encrypt}_r(p)\rangle$
    for $i \leftarrow 1, \ldots, |\mathcal{A}|$ do                ▷ Scan each clause
        Parse $C_i = \langle e_1, \ldots, e_{l(i)}\rangle$
        for $q = 1, \ldots, l(i)$ do                ▷ Seach decryption key
            for all $k_w \in \mathcal{I}$ do
                $s_q \leftarrow \texttt{Decrypt}_{\hat{k}_w}(e_q)$
                if $s_q \neq \perp$ then
                    break
                end if
            end for
        end for
        if (Correctly decrypted each share in $C_i$) then
            $r \leftarrow \bigoplus_{q=1}^{l(i)} s_q$
            return $r$
        end if
    end for
    return $\perp$
end function
```

7.4 Security discussion

The formula transformation *partially* solves the issues related to the publicity of the access structure since every access structure is now defined by a DNF formula. On the other hand, an adversary obtains information about the number of clauses in the formula and their size. Furthermore, an adversary can infer information about the software packages she has locally installed, e.g., if the key k_a correctly decrypts a literal in a clause then the adversary knows that current package depends on feature f_a. We notice that this transformation tends to gain privacy by loosing efficiency. Indeed the transformation from an access structure to its corresponding basis might bring an exponential blow-up in the policy size, e.g., in the case of threshold access structure. Furthermore, the anonimization of package names forces the user to linearly search correct keys instead of directly accessing them.

8 Performance Evaluation

In this section we present the experimental evaluation we have carried out for testing the performance of the proposed software installation system. Our system has an impact on two operations in the package management lifecycle, namely, package creation and installation. Both operations consists of two separate phases.

Key processing. During the first phase the system either shares the encryption key according to the installation policy or reconstructs it, given the encrypted shares, the installation policy and the feature keys. The performance of this phase heavily depends on the performance of our scheme. Indeed, the amount of additional information to be stored and the amount of time needed to run this phase depend on the protocol type and on the access structure to be implemented.

Package processing. During this phase the package is either created and encrypted or decrypted and installed. The (d)encryption operation only depends on the size of the package. We notice that the package contains a specification of the installation policy, whose size depends on the specific protocol distribution protocol we use. On the other hand, as we will see, the size of this specification is negligible w.r.t. the size of the software contained in the package. For this reason, the time needed to execute this phase is *independent* from our protocols and we do not consider this phase in our evaluation.

8.1 The experimental setup

For the sake of an experimental evaluation, we tested the proposed software update system by extending the RPM Package Management System, one of the most deployed PMSes. Our experimental test set consists of all the packages contained in the following Linux Distributions: CentOS, Red Hat Enterprise Linux (RHEL), OpenSuse and Fedora. We have considered the installation policy in each package and classified it according to the measures we discuss below. We have considered two extreme cases. The former case corresponds to the point of view of a software vendor that needs to encode *all the packages in a given distribution*, by making each package backward compatible with some of its previous versions. The latter one, corresponds to the case in which the vendor needs to prepare a *core* distribution, i.e., a minimal set of packages that guarantee the system to operate properly.

The experiments have been carried out on a dedicated desktop PC powered by an Intel® Core™i5 CPU M460 at 2.53 GHz x4, equipped with 5GB Ram and a 1TB HD and running a Linux Ubuntu 15.10 32-bit. All policy-size experiments were conducted by ad-hoc Python 2.7 scripts leveraging python rpmlib version 4.12, while time-measurements were conducted by a Java application.

8.1.1 RPM file format

Each RPM package is encoded in a single file, formatted according to the rpm format. Since its very first versions, RPM was designed to allow each package to (a) describe the set capabilities it requires to be executed, (b) list the set of capabilities it provides and (c) store and retrieve such capabilities from the RPM database.

The process implemented by the RPM package manager to build an rpm file first analyses the list of files to be packaged. It constructs the list of capabilities provided by the package and, most importantly, the list of capabilities required by the package. The required capabilities that correspond to dependencies are classified by RPM into *automatic* and *manual* ones. Automatic dependencies correspond to the set of shared libraries that are required by the current package and are computed by the package manager using the `ldd` command. Since *all* automatic dependencies are required to properly run the current package, their associated installation policy corresponds to a single-clause DNF formula.

Manual dependencies correspond to specific capabilities that the package creator might require. Examples of such capabilities are the requirements for a specific package or for a specific set of versions of a given package. Versions can also be specified by using the operators $<, \leq, =, >$ or $\geq$, e.g., `requires: foo >= 1.5`.

RPM format allows the specification of features that conflict with the current one, i.e., package that cannot be installed simultaneously with the current package. Conflicts correspond to the logical complement of required features and are represented by non-monotone formulas. The management of non-monotone formulas is currently under investigations.

8.1.2 RPM installation policies and boolean formulae

If we consider the boolean formulae that can be constructed by the RPM package manager, we can state that they only consist of conjunctions of the following types of formulas:

type-a. Automatic dependencies correspond to a formula consisting of exactly one DNF-clause that contains one literal for each required feature f_i;

type-b. For each additional vendor requirement, a formula consisting of the disjunction of literals associated to the features $f'_{i,v_{i,1}}, \ldots, f'_{i,v_{i,k_i}}$ corresponding to the different versions of the required feature f'_i.

In general, let us assume the (only) type-a clause consists of k literals. Furthermore, let c be the number of type-b clauses in the installation policy and let us denote by k_i, $i = 1, 2, \ldots, c$ the number of literals in the i-th clause. The installation policy generated by the RPM manager can be described by the following boolean formula:

$$\hat{\phi} = \bigwedge_{i=1}^{k} f_i \wedge \bigwedge_{i=1}^{c} \left(\bigvee_{j=1}^{k_i} f'_{i,v_{i,j}} \right) \tag{1}$$

This formula can be used as is in the protocols described by Algorithms 2 and 3, i.e., when the installation policy is public or when we do require the partial hiding of the policy as described in Section 7.2. Notice that it is possible to store this formula by using space $|\hat{\phi}| = O(k + k_1 + \ldots + k_c)$.

In case of fully-hiding protocols, described by Algorithms 4 and 5, we need to convert the installation policy in DNF, that corresponds to the following formula $\phi = \bigvee_{j_1=1}^{k_1} \ldots \bigvee_{j_c=1}^{k_c} (\bigwedge_{i=1}^{k} f_i \wedge f'_{1,v_{1,j_1}} \wedge \ldots \wedge f'_{c,v_{c,j_c}})$

The formula consists of $\prod_{i=1}^{c} k_i$ clauses, each with exactly $k + c$ literals. Thus the size of the formula can be computed as $|\phi| = O\left((k + c) \prod_{i=1}^{c} k_i\right)$.

8.1.3 Estimating the policy size

As stated before, $|\hat{\phi}|$ and $|\phi|$ express the number of literals in the formula associated to the installation policy. Specifically, the former corresponds to the public or partially-hidden policy, while the latter to the fully-hidden policy. For each literal we need to store a constant number of bytes, namely the pair (feature name, encrypted share). Thus the total size of the formula is linear in the number of literals.

In our experiments we have evaluated the above formulas in real-life operational settings for the test distributions CentOS, Red Hat Enterprise Linux (RHEL), OpenSuse and Fedora. More precisely, for each test distribution we have evaluated the values of k, c and $k_1, \ldots, k_c$ by considering either 2 or 4 consecutive versions of each test distribution, as listed in Table 1.

To do so, we have first constructed a database containing the list of all pairs *(Feature, Feature Version)*, that are provided by all the packages in all the versions of a given distributions in the test set. Given such a database, for each (distinct) package, it is possible to obtain the value of k as the number of required features for which a single version, compatible with the installation policy, exists in the database. Similarly, the value of c is obtained by counting the number of required features for which multiple versions, compatible with the installation policy, exist in the database. Finally, the values $k_1, \ldots, k_c$ is computed by counting the number of different versions of each required features that is present in the database.

Let us consider, for example, the Fedora Server distribution. We have constructed the feature-version database by listing all features contained in the Fedora distributions $20, 21, 22$ and 23. At this point, for each (distinct) package in any of the above distributions, we have computed the value c by counting the number manual requirements listed in the installation policy. Furthermore, for each such requirement, we have computed k_i by counting the number of features in the database that satisfy the requirement.

For example, assume that featureA version 3.2.9 is contained in Fedora Server 20 and 21. The same feature is contained in version 3.3.0 in Fedora Server 22 and versions 3.3.1 in Fedora Server 23. Furthermore, let us assume that featureA is required in its version `<=3.3.1` in some package in Fedora Server 23. Thus the list of all pairs in the database that are compatible with this requirement is (featureA, 3.2.9), (featureA, 3.3.0) and (featureA, 3.3.1), that has cardinality $k_i = 3$.

8.1.4 Formula size in the protocol with public or partially-hidden policy

In order to guarantee the correctness of the protocol defined by Algorithm 3, the package needs to store (a) a specification of the formula defined by Equation 1 and (b) the encrypted shares for pairs (feature, version). In this case the formula can be succinctly represented by using $k+k_1+\ldots+k_c$ pairs (feature, version). Furthermore, the package will contain $k + k_1 + \ldots + k_c$ encrypted shares, again, one for each pairs (feature, version) contained in the installation policy,

Table 2 contain the measurements over the different distributions in test set. Specifically, Table 2 contains the maximum, average number and the percentage of packages with a given range of required for the parameters c, k, i.e., the number of required features for which one or multiple versions exist in the feature database. Furthermore it contains the percentage of the number of clauses in the formula

TABLE 1
Experimental set up.

Distribution	Experiment	Full Distributions				Core Distributions			
Name	Acronym	Ver.	#Packages	# Provides	#Requires	Ver.	#Packages	# Provides	#Requires
CentOS (Minimal)	CentOS-M	6-7	574	3435	8044	7	357	410	655
CentOS (Complete)	CentOS-C	4-5-6-7	20737	122312	261896	7	169	673	823
Fedora	Fedora	20-21-22-23	11298	200467	142640	23	145	264	541
OpenSuse	OpenSuse	11-12-13-13.2	19444	175181	229503	13.2	1267	6558	9002
RedHat Enterprise	RHEL	4-5-6-7	13292	93019	196994	7	166	678	837

whose size lays in the given range. From the data reported in such table, we can notice that, for the vast majority of the packages in all the distributions' set, the size of the formula that corresponds to the installation policy is small, i.e., below 50 literals. In the worst case, the number of literals is below 531, that corresponds to a space of few kilobytes on disk.

Let us consider the time needed to execute the key sharing/recovery phase. In the case the policy is public, the vendor/user needs to execute a number of (symmetric) share encryptions (resp., decryptions) that is upper bounded by the number of literals in the formula. Since such a number is upper bounded by few hundreds, such key sharing/recovery phase takes few milliseconds.

The situation becomes slightly more complex when the policy is required to be partially hidden. In this case, the encrypted share associated to each literal in the formula has to be decrypted by using each key in the local database. As previously highlighted, in practical applications, the number of installed features remains well below the number of all available features. Analogously, the amount of space to store the feature keys along with the other metadata maintained by the package management system is affordable as it is few bytes for each installed feature. In any case, these phases take at most few seconds.

8.1.5 Formula size in the protocol with fully-hidden policy

The same evaluation has been executed for the space required to store policies in DNF. In this case, the formula cannot be stored in a compact way since each clause in the DNF has to be written in the installation policy. The expansion factor, defined by $|\phi|$, has been computed for each package in each target distribution. Table 3 contains the cumulative percentage for the distribution of the expansion factors. From these results, it is clear that the additional security requirement of fully hiding the installation policy comes to the cost of a blow-up of the space needed to store the installation policy, and, consequently, of the time needed to share/recovery encryption keys that, in some cases, may be relevant.

8.1.6 Performance on core installations

The results reported in the previous paragraphs have been obtained by considering the installation of *all the packages* in the database. Clearly, in highly sensitive systems, it is never the case that the administrator installs unused and potentially dangerous packages. For this reason we have considered, for each distribution, the set of packages that identify the *core* installation for a *single* distribution version. A *core* installation is the minimal sets of packages for which (a) all dependencies are met, i.e., there exist no dependency from a feature provided by a package outside the set and (b) the installed system is fully functional. Notice that the CentOS *minimal* distribution strictly contains its core installation, as it includes packages that ease the usage of the system but that are not required for the system to work properly (*e.g.*, the X/Windows environment and the LibreOffice productivity suite). Table 1 reports the number of package, provided features and required features for core installations of the distributions in the test set. Such values are one or two orders of magnitude smaller than the values corresponding to the full distributions case. Tables 2 and 3 report the formula size for the public/partially hidden case and the fully hidden one.

8.1.7 Key Processing Time

In this section we briefly report the simulation of key reconstruction algorithms presented in this paper. Specifically, we have only computed the time for (the appropriate number of) share decryptions while we have not reported the time required to query the local inventory, to iterate over keys in therein contained or to run the secret sharing reconstruction algorithm.

We start by defining a set of parameters, $|\mathcal{I}|$, k, c and k_i. As a first step we generate a database of random feature keys of size $|\mathcal{I}|$. We then compute the proper number of shares, $|\phi|$ or $|\hat{\phi}|$, that correspond to an installation policy with parameters k, c and $k_1 = \ldots = k_c = k_i$, and encrypt them with the corresponding keys. Given this set of encrypted shares, we run the reconstruction algorithms presented in the paper.

Table 4 reports the reconstruction time required by protocols for public or partially hidden policies. We have chosen to test policy sizes $\hat{\phi}$ of size approximately 50 and 100 as these are the limits of the cumulative percentage reported in Table 2. Furthermore, we have chosen the number of keys in the local inventory in the range 10^3, 10^5 since these are the orders of magnitude for inventories reported in Table 1. Clearly, the time needed in case of public policies does not depend on $|\mathcal{I}|$ since, in this case, given the feature name, the algorithm queries the database and retrieves the correct key. In all cases, few milliseconds are sufficient to correctly decrypt all the shares in the policy. In the case of partially hidden policies, instead, the algorithm needs to iterate over the keys in the inventory. We have considered the worst case scenario in which (a) automatic dependencies always correspond to the last keys in the iteration (i.e., for each

TABLE 2
Cumulative percentage for the expansion parameters in the public or partially-hidden policies for Full and Core Distributions.

Exp. Acronym	Full Distributions										Core Distributions		
	c			k			$\|\hat{\phi}\| = k + k_1 + \ldots + k_c$				$\|\hat{\phi}\| = k + k_1 + \ldots + k_c$		
	Av.	Max	0-10	Av.	Max	0-100	Av.	Max	0-50	0-100	Av.	Max	0-50
CentOS-M	0,90	8	100,00%	13,10	80	100,00%	15,14	87	97,91%	100,00%	16,60	62	98,04%
CentOS-C	1,07	32	99,75%	11,56	232	99,67%	17,28	531	94,12%	99,09%	12,37	62	98,82%
Fedora	0,85	38	99,81%	11,78	319	99,46%	16,46	437	95,18%	99,16%	17,88	50	100,00%
OpenSuse	0,94	44	99,13%	10,86	223	99,76%	15,64	252	95,04%	99,41%	15,17	74	98,18%
RHEL	1,06	32	99,71%	13,71	232	99,59%	19,75	522	92,70%	98,84%	12,63	62	98,80%

TABLE 3
Cumulative percentage for the expansion factor for the installation policy size in the fully-hidden policies for Full and Core Distributions.

Exp. Acronym	Full Distributions				Core Distributions			
	$0 - 10^3$	$0 - 10^4$	$0 - 10^5$	$0 - 10^6$	$0 - 10^3$	$0 - 10^4$	$0 - 10^5$	$0 - 10^6$
CentOS-M	95,64%	98,95%	99,83%	99,83%	95,24%	98,88%	99,72%	100,00%
CentOS-C	84,88%	91,22%	94,50%	96,25%	91,12%	97,63%	98,22%	99,41%
Fedora	87,27%	93,34%	96,19%	97,38%	89,66%	97,24%	97,24%	100,00%
OpenSuse	87,44%	92,99%	96,03%	97,63%	80,90%	90,84%	95,58%	96,76%
RHEL	82,98%	89,78%	93,37%	95,46%	98,19%	98,80%	99,40%	100,00%

such share, the algorithm executes exactly $|\mathcal{I}|$ decryptions) and (b) each manual dependency is fails $k_i/2$ times, (i.e., a successful share decryption follows $|\mathcal{I}|k_i/2$ unsuccessful ones). Also in this case, the timings reported in Table 4 show that the reconstruction times are affordable even in the case of complex policies and huge local inventory.

Table 4 reports the reconstruction time required by protocols for fully hidden policies. In this case we have focused our attention to the case of core distributions.

TABLE 4
Public and partially hidden policy key reconstruction times

Parameters			Running time (ms)	
$\|\mathcal{I}\|$	k	$\|\hat{\phi}\|$	Public	Part. hidden
1000	10	50	2	544
1000	100	140	10	1130
10000	10	50	2	2254
10000	100	140	11	7437
100000	10	50	2	17340
100000	100	140	9	70462

TABLE 5
Fully hidden policy key reconstruction times

Parameters			Running time (ms)			
			$k = 10$		$k = 50$	
$\|\mathcal{I}\|$	c	k_i	$\|\phi\|$	time	$\|\phi\|$	time
100	2	5	300	197	1300	405
100	2	10	1200	367	5200	667
100	4	5	8750	1145	33750	2791
100	4	10	140000	12282	540000	40060
1000	2	5	300	751	1300	1652
1000	2	10	1200	1722	5200	4523
1000	4	5	8750	8646	33750	26438
1000	4	10	140000	126112	540000	385962

9 Conclusions and Open Problems

In this paper we have presented protocols for ensuring strong dependency enforcement during the software installation process. Our protocols improve over previous solutions for a number of reasons. In our model, we explicitly consider multiple independent software issuers that are not required to cooperate in any way. The enforcement of dependencies is strong in the sense that the package is distributed in an encrypted form and it can be decrypted by the user if and only if the device meets the issuer-defined installation policy. Finally, the update of the device's key material is executed locally without any interaction with the issuer, solving issues related to the inefficiency of interactive protocols or profiling of user devices. Furthermore, we have presented variants of our protocols in which the description of dependencies can be kept either partially or fully hidden to an adversary by sacrifying, in part, the efficiency of the solution. We have presented and analyzed the protocols from a theoretical point of view and experimentally evaluated their performance.

Our protocols are based on secret sharing schemes that only allow the management of monotone installation policies. We are currently evaluating techniques to allow the management of non-monotone policies, *i.e.*, the ones that allow the installation of a package only if some feature is *not* installed on the system.

References

[1] L. Catuogno, C. Galdi, and G. Persiano, "Guaranteeing dependency enforcement in software updates," in *Secure IT Systems, 20th Nordic Conference, NordSec 2015, Stockholm, Sweden, October 19-21, 2015, Proceedings*, 2015, pp. 205–212.
[2] T. Dumitraş, S. Kavulya, and P. Narasimhan, "A fault model for upgrades in distributed systems (cmu-pdl-08-115)," 2008.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.

[3] A. D. Rubin, "Trusted distribution of software over the internet," in *1995 Symposium on Network and Distributed System Security, (S)NDSS '95, San Diego, California, February 16-17, 1995*, 1995, pp. 47–53.

[4] J. Samuel and J. Cappos, "Package managers still vulnerable: How to protect your systems," *login: Usenix Magazine, Feb*, vol. 34, no. 1, pp. 7–15, 2009.

[5] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman, "A look in the mirror: Attacks on package managers," in *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 565–574.

[6] A. Bellissimo, J. Burgess, and K. Fu, "Secure software updates: Disappointments and new challenges." in *HotSec*, 2006.

[7] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine, "Survivable key compromise in software update systems," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 61–72.

[8] L. Catuogno, R. Gassirà, M. Masullo, and I. Visconti, "Smartk: Smart cards in operating systems at kernel level," *Information Security Technical Report*, vol. 17, no. 3, pp. 93 – 104, 2013, security and Privacy for Digital Ecosystems.

[9] Microsoft Corp., "Introduction to code signing," https://msdn.microsoft.com/en-us/library/ms537361.aspx, Microsoft Development Network.

[10] AA. VV., "Signing your applications," http://developer.android.com/tools/publishing/app-signing.html, 2015.

[11] J. Cappos, "Avoiding theoretical optimality to efficiently and privately retrieve security updates," in *Financial Cryptography and Data Security - 17th International Conference, FC 2013*, 2013, pp. 386–394.

[12] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," *J. ACM*, vol. 45, no. 6, pp. 965–981, Nov. 1998.

[13] E. Dolstra, M. De Jonge, and E. Visser, "Nix: A safe and policy-free system for software deployment." in *LISA*, vol. 4, 2004, pp. 79–92.

[14] E. Foster-Johnson, "Red hat rpm guide," 2003.

[15] J. Hart and J. D'Amelia, "An analysis of RPM validation drift." in *LISA*, vol. 2, 2002, pp. 155–166.

[16] S. Neuhaus and T. Zimmermann, "The beauty and the beast: Vulnerabilities in Red Hat's packages." in *USENIX Annual Technical Conference*, 2009.

[17] M. Ambrosin, C. Busold, M. Conti, A.-R. Sadeghi, and M. Schunter, "Updaticator: Updating billions of devices by an efficient, scalable and secure software update distribution over untrusted cache-enabled networks." in *ESORICS (2014)*, 2014, pp. 76–93.

[18] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-policy attribute-based encryption," in *Security and Privacy, 2007. SP'07. IEEE Symposium on*. IEEE, 2007, pp. 321–334.

[19] A. Sahai and B. Waters, "Fuzzy identity-based encryption," in *Advances in Cryptology, EUROCRYPT 2005*, ser. Lecture Notes in Computer Science, R. Cramer, Ed. Springer Berlin Heidelberg, 2005, vol. 3494, pp. 457–473. [Online]. Available: http://dx.doi.org/10.1007/11426639_27

[20] R. Ostrovsky, A. Sahai, and B. Waters, "Attribute-based encryption with non-monotonic access structures," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 195–203.

[21] S. Müller, S. Katzenbeisser, and C. Eckert, *Distributed Attribute-Based Encryption*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 20–36. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00730-9_2

[22] M. Chase, "Multi-authority attribute based encryption," in *Proceedings of the 4th Conference on Theory of Cryptography*, ser. TCC'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 515–534.

[23] J. Han, W. Susilo, Y. Mu, J. Zhou, and M. H. Au, "PPDCP-ABE: privacy-preserving decentralized ciphertext-policy attribute-based encryption," in *Computer Security - ESORICS 2014 - 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II*, 2014, pp. 73–90.

[24] GlobalPlatform, "Tee system architecture v1.0," http://www.globalplatform.org, December 2011.

[25] L. Catuogno and C. Galdi, "Ensuring application integrity: A survey on techniques and tools," in *2015 9th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, July 2015, pp. 192–199.

[26] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn, "Design and implementation of a TCG-based integrity measurement architecture." in *USENIX Security Symposium*, vol. 13, 2004, pp. 223–238.

[27] UEFI, "Unified extensible firmware interface specification," http://www.uefi.org/sites/default/files/resources/UEFI_Spec_2_7.pdf, pp. 1967–2000, May 2017.

[28] GLobalPlatform Inc., "TEE System Architecture (version 1.1)," January 2017, doc. ref.: GPD_SPE_009.

[29] J. C. Benaloh and J. Leichter, "Generalized secret sharing and monotone functions," in *Advances in Cryptology - CRYPTO '88, 8th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1988, Proceedings*, 1988, pp. 27–35.

[30] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[31] M. Ito, A. Saito, and T. Nishizeki, "Secret sharing scheme realizing general access structure," *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, vol. 72, no. 9, pp. 56–64, 1989.

[32] J. Benaloh and J. Leichter, "Generalized secret sharing and monotone functions," in *Proceedings on Advances in cryptology*. Springer-Verlag New York, Inc., 1990, pp. 27–35.

[33] D. R. Stinson, "New general lower bounds on the information rate of secret sharing schemes," in *Advances in Cryptology (CRYPTO 92)*. Springer, 1993, pp. 168–182.

[34] G. D. Crescenzo and C. Galdi, "Hypergraph decomposition and secret sharing," *Discrete Applied Mathematics*, vol. 157, no. 5, pp. 928–946, 2009.

[35] Common Criteria Sponsoring Organizations, "Common criteria for information technology security evaluation part 1: Introduction and general model, version 3.1 rev 4,," September 2012.

Luigi Catuogno is network administrator at the Department of Computer Science of the University of Salerno, where he got his Laurea degree (M.Sc. equivalent) in 1999, his PhD in Computer Science in 2004 and a post-doc fellowship in 2009. He has been visiting student at New York University in 2002 and research assistant at Ruhr University of Bochum (Germany) in 2009. His research focuses on system security and privacy enhancing technologies.

Clemente Galdi got a Laurea cum laude and PhD in Computer Science from Università degli Studi di Salerno in 1997 and 2002. He has been visiting researcher at Telcordia Technologies and DIMACS, New Jersey, USA. He has been post-doctoral fellow at Computer Technology Institute and University of Patras (Greece) (2001-2004), at Università di Salerno (2004-2006). Since April 2006, he is Assistant Professor at the Università di Napoli. His fields of interests are Data Security, Cryptography and Algorithm Design.

Giuseppe Persiano got a *Laurea in Scienze dell'Informazione* at the Università di Salerno in 1986 ans a PhD in Computer Science from the Harvard University (Cambridge, MA, USA) in 1994. He was associate professor at the Università di Catania (1992-1994) and, since 1994 he is professor of Computer Science at Università di Salerno. He has visited several research institutions and universities in USA (DIMACS, Univ. of California at Berkeley, Univ. of California at Los Angeles, Google) and Europe (Univ. of Paderborn, Univ. of Patras). His research focuses on Cryptography and algorithmic aspects of Game Theory and Microeconomic.