International Journal of Information Security manuscript No.
(will be inserted by the editor)

PDGuard: An Architecture for the Control and Secure Processing of Personal Data

Dimitris Mitropoulos · Thodoris Sotiropoulos · Nikos Koutsovasilis · Diomidis Spinellis

Received: date / Accepted: date

Abstract Online personal data are rarely, if ever, effectively controlled by the users they concern. Worse, as demonstrated by the numerous leaks reported each week, the organizations that store and process them fail to adequately safeguard the required confidentiality. In this paper we propose PDGuard, a framework that defines, prototypes, and demonstrates an architecture and an implementation that address both problems. In the context of PDGuard, personal data are always stored encrypted as opaque objects. Processing them can only be performed through the PDGuard Application Programming Interface (API), under data and action-specific authorizations supplied online by third-party agents. Through these agents end-users can easily and reliably authorize and audit how organizations use their personal data. A static verifier can be employed to identify accidental API misuses. Following a security by design approach, PDGuard changes the problem of personal data management from the, apparently, intractable problem of supervising processes, operations, per-

Dimitris Mitropoulos, Thodoris Sotiropoulos, Nikos Koutsovasilis, and Diomidis Spinellis. PDGuard: An architecture for the control and secure processing of personal data. *International Journal of Information Security*, September 2019.

This is a machine-readable rendering of a working paper draft that led to a publication. The publication should always be cited in preference to this draft using the reference in the previous footnote. The final publication is available at Springer via `http://dx.doi.org/10.1007/s10207-019-00468-5`.

Dimitris Mitropoulos, Thodoris Sotiropoulos, Diomidis Spinellis
Department of Management Science and Technology
Athens University of Economics and Business
E-mail: {dimitro, theosotr, dds}@aueb.gr
Partial work by Dimitris Mitropoulos was done while at the
Department of Computer Science of
Columbia University in the City of New York

Nikos Koutsovasilis
Department of Informatics and Telecommunications
University of Athens
E-mail: sdi1500076@di.uoa.gr

sonnel, and a large software stack to that of auditing the applications that use the framework for compliance. We demonstrate the framework's applicability through a reference implementation, by building a PDGuard-based e-shop, and by integrating PDGuard into the *The Guardian* newspaper's website identity application.

1 Introduction

The protection of personal data in the digital world is clearly insufficient. Numerous data breaches, which regularly come to light, indicate that organizations entrusted with personal data fail to secure them effectively against internal and external threats [33, 40, 25]. Personal data breaches can affect millions (e.g. when credit card details are obtained from a hacked database) or particular individuals (e.g. when an officer investigates his girlfriend's activities on digital government records). Unauthorized disclosure of personal data violates privacy rights, breeds distrust between organizations holding the data and the people associated with it, burdens and distracts both parties, and interferes with the functioning of the digital economy. In this paper we introduce PDGuard, a framework that enables the control and secure processing of personal data (i.e. protecting them from potential misuses).

Personal data is any information that can be used to identify, contact, or locate a person in a specific context [8, 45, 22, 36]. In this paper we use the terms *data subject* to refer to a person associated with personal data, and *data controller* to refer to a public or private organization holding and processing those data. The protection of personal data is insufficient, because the current framework for its protection is opaque and ineffective.

Motivation The protection of personal data is **opaque** on four layers: security policies, their implementation, the links between policies and enforcement mechanisms, and authorized and unauthorized uses of personal data. Many of these elements are not visible to outsiders. When they are visible they may be inscrutable (as is often the case with security policies), too coarse grained (as happens with the obligatory reporting of data breaches), and expensive to interpret, audit, and distill into actionable information. Adding insult to injury, visibility is often limited to internal auditors and regulators, with data subjects typically kept in the dark regarding the handling of their personal data.

Entire books can (and have) been written on why the protection of personal data is **ineffective** [21, 56, 14]. In the context of our work, the reasons can be summarized as follows. First, excessive trust is placed on the data controller, which results in valuable data being protected by a single, often insufficiently resourced entity. Then, the manual enforcement of security policies is difficult and costly to standardize and verify, and also varies in effectiveness according to the training, motivation, and ability of the personnel that applies them. Also, the technical means of policy enforcement are typically applied on the (too many) leaf nodes of a vulnerability tree (e.g. by disabling storage devices on every USB port). This results in a large, porous, and insufficiently understood attack surface. Furthermore, the protection's opaqueness we described, hinders informed consumer choice, democratic accountability, regulation (self- and by outsiders), and the functioning of market-based incentives.

Currently, any **control** data subjects may have over their personal data is often too coarse (typically an all or nothing proposition), confusing (as it varies with each data controller or even between applications), and manually applied according to the willingness or even whim of each data controller. Even though there are many steps towards a more user-centric privacy identity management approaches [2, 1, 23, 17, 55], in many countries, data controllers handle personal data as they see fit, while numerous online businesses gather, analyze and use them at best within the limits of an inscrutable take-it-or-leave-it personal data policy, which very few subjects bother to read [11, 30, 39, 29].

Overview PDGuard is an IT system architecture backed by an application programming interface (API) and an open-source software reference implementation that aims to provide security, transparency, and effective control in the handling of personal data. Within the context of PDGuard, personal data are always stored encrypted as an opaque object. Decryption can only be performed through the PDGuard API, under data and action-specific authorizations supplied by a third-party, an *escrow agent*, which is an entity trusted by both the data subject and the controller. By interacting with escrow agents, data subjects can reliably authorize and audit how data controllers use their personal data.

When data subjects establish a relationship with a data controller they supply the controller with the address of the escrow agent of their choice for the specific relationship. For data subjects not interested to setup a relationship with an external escrow agent, PDGuard uses a default *internal* escrow agent, which offers the same functionality and implements the data controller's personal data policy. Data subjects may decide to choose diverse *external* escrow agents so that they can apply different policies on different transactions (according e.g. to the type of the personal data and the applications that request them). The data controller will send to the escrow agent data storage and access requests according to its business needs, its personal data protection policy, as well as legal and regulatory requirements. In turn, data subjects will authorize the requests as they see fit.

The data controller software applications perform data encryption and decryption on-demand with *keys* supplied each time through the escrow agent's authorization service, in response to *authenticated entity* requests. In essence, we take a "defense-in-depth" [64, 19] approach by storing encrypted entities and corresponding keys in different locations. Each escrow agent allows data subjects to associate permissions with specific *data types*, *data uses*, and more. For example, a data subject, Mary, can allow a data controller, Acme, to use her postal address (data type) for labeling (data use) a gadget she ordered, but not for sending her advertisements (data use). The data regarding all authorizations granted by an agent are made available to the data subjects, so that they can review them or revoke future uses. Dedicated secure key servers and the long-term caching of keys limit the performance impact of key distribution.

A *static code verification tool* can be employed to identify accidental framework misuses within an application's software code. In particular, developers can review data controller applications and check if the decrypted data retrieved through the PDGuard API, are used as intended (recall that permissions are associated with data uses).

Contributions PDGuard provides a way for data subjects to control their online personal data, and *at the same time* protect the data from diverse threats associated with them (see Section 2.2). There have been several approaches to provide the former or latter functionality, (see Section 7) but, to the best of our knowledge, no approaches combine the two and exploit the associated synergies. Specifically, by combining the two, PDGuard reduces the complexity, the implementation and the verification effort over a system that provides either the former or the latter functionality.

PDGuard addresses the opaqueness in the handling of personal data, by defining an API and an escrow agent that data controllers shall use for processing them. This provides data subjects with a clear unified view of the security policy and its implementation through e.g. the escrow agent's web user interface. It also allows data subjects to review how their personal data were used. Furthermore, adherence to the PDGuard framework can be audited and published, both at the level of data controllers and at the level of individual software applications.

Following the "security by design" [66, 31, 12, 41, 61] approach, PDGuard improves the effectiveness of personal data protection in the following ways:

- It *decentralizes* trust and control by having numerous escrow agents of (hopefully) diverse implementations control the decryption keys. This empowers data subjects with fine-grained control of how their personal data are used.
- It *replaces* a patchwork of manually enforced security policies with a standardized API that can be uniformly applied, monitored, and audited across all data controllers and software applications.
- It *reduces* the attack surface and the risk of individual vulnerabilities of the systems it is deployed on, through the decryption of personal data at the time of their use. Note that, the escrow agent generates keys by taking into account different data uses.
- It *increases* transparency in the handling of personal data. This allows data subjects to choose among data controllers based on the concrete protection of their personal data that they offer (adoption of PDGuard and actual data use), provides regulators with an easy way to establish whether personal data is effectively protected, and allows market-based mechanisms to spread adoption of competing PDGuard implementations, escrow agents, and auditing services.

2 Background

PDGuard brings together four entities under a threat model where data controllers and users wish to safeguard their data against third party malicious attacks. These entities are aligned with the ones that have been introduced in the context of the personal data regulation of the European Union (i.e. the General Data Protection Regulation (GDPR) [8]. Notably, GDPR recommends that the development of applications incorporating personal data should follow a "security by design" approach, which is compatible with PDGuard.

2.1 Participating Entities

The entities that exist in the context of PDGuard are the following.

1. *Data Subject*: The individual associated with personal data. This is the entity whose privacy concerns restrict the permitted data processing operations. In some cases, we refer to data subjects simply as *users*.
2. *Data Controller*: A public or private organization that processes personal data associated with data subjects. Examples include providers of e-commerce, e-government, social networks, and other services.
3. *Data Controller Applications*: The data controller's software systems that process personal data. Such systems may handle interactive transactions, back-office fulfilment, reporting, analytics, and relationship management.
4. *Escrow Agent*: A third party software application and service trusted by both the data subject and the data controller to facilitate and monitor the compliance of the data controller's processing activities according to the data subject's instructions.

Note that in GDPR the entities that process personal data on behalf of the controller are called "Data Processors". In a situation like this, the applications running on behalf of the data processor, should be treated in the same manner as the data controller applications.

2.2 Threat Model

We define the threat model under which PDGuard operates in terms of trust, addressed and residual threats. Details on how threats are handled are given in Section 5.4, "Security Analysis".

In the PDGuard framework (qualified) **trust** is placed on the participating entities, namely the data controller, the data subject, and the escrow agent. Specifically, we expect that data controller applications aim to respect the authorizations that a data subject sets through an escrow agent to prevent data misuse. We also assume that the developers of the various applications will strive to use correctly the PDGuard API. In particular, the data controller's applications must: (1) handle all personal data through the PDGuard API, (2) issue decryption requests with correctly specified data type and data use fields, (3) avoid storing personal data locally (in memory or on non-volatile storage) after using them, and (4) never use data or keys obtained for one purpose for a different one (e.g. forward the data subject's contact phone number to a telemarketer).

The PDGuard framework **addresses diverse internal and external threats**. Malicious entities aim to exploit the data for purposes different than those allowed by the law, the desire of the data subjects, and possible agreement between data subjects and controllers. External threats include the unauthorized access to the data controller infrastructure which would allow an outsider to retrieve data, the misuse of personal data by advertising networks, data brokers, online tracking, and more. In addition, they may include the exploitation of an application by hackers, phishing, and malware. An internal threat involves an employee's behavior that puts personal data at risk, despite the security policies and mechanisms that the data controller utilizes. Transferring files between work and personal computers, unauthorized application use, and sharing work devices with third-parties without supervision, are some indicative cases of such behaviors.

Although malicious data controllers are outside the scope of PDGuard, within its context misrepresentation and overreaching are controlled, negligence is mitigated, and adherence can be effectively verified through the accompanying static code verification mechanism (see Section 3.4). Thus, regarding data controllers, the PDGuard stance can be summed up as "trust, but verify". This improves upon the current state of the art in this area, which is mainly based on trust.

Nevertheless, a number of **residual threats** remain. Notably, in an environment with insufficient separation of roles, lax auditing, or sloppy key management, unauthorized administrators or developers might be able to leverage the PDGuard API to process personal data. Furthermore, the PDGuard architecture does not aim to prevent all attacks that target data controller applications containing software vulnerabilities. There are specific web application attack vectors that will fail because of the framework's design. We examine such cases in our "Security Analysis" section (5.4).

2.3 Use Case Scenario

Consider the following scenario, which we will use as a reference throughout the rest of this paper. Alice visits a news and media website on a daily basis. She decides to sign up and create a personal account to interact with other readers in the website's forum and receive email notifications with weekly digests. Also, she wants to receive the printed version of a magazine, published by the website for two years on a monthly basis. During her registration, Alice submits the following credentials: name, surname, password, email, credit card number and street address.

Alice wants only her name to be displayed on the forum's pages and does not want to receive emails from third party organizations screened by the website. In addition, she wants to have the ability to update her address through the website, and allow it to access her credit card automatically when her magazine subscription ends.

Immediately, we can identify Alice as the data subject and the website as the data controller. We can assume that the website runs diverse applications, including the forum and the service where users can handle and update their

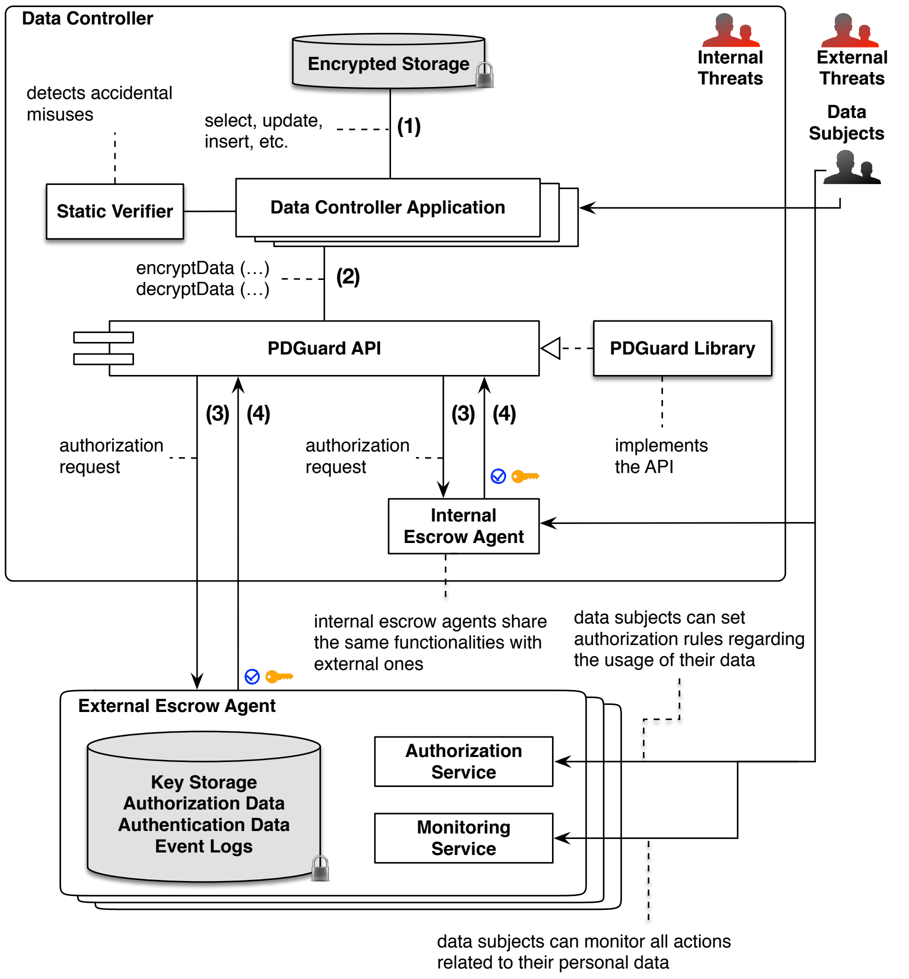


Fig. 1: **A data processing system supported by PDGuard**. Personal data are stored encrypted as opaque objects. Every time a DB action (**1**) involving personal data is performed, data controller applications invoke the PDGuard API (**2**). In turn, the API sends authorization requests (**3**) to the corresponding escrow agent (can be either external or internal). If a request conforms to the rules set by the data subject, the agent responds with the required decryption key (**4**).

profiles. We can also assume that Alice has already chosen an escrow agent through which she can specify the rules regarding the use of her personal data.

3 Architecture

Figure 1 presents a data processing system supported by the PDGuard framework. In this context, all personal data are stored encrypted as opaque objects in the data controller. All data controller applications that perform DB actionsinvolving personal data, have to use the PDGuard API to perform the corresponding encryption or decryption. When an API call is invoked, an authorization request is sent to the corresponding escrow agent that maintains the encryption key (an agent can be either external or internal). If a request conforms to the rules set by the data subject, the agent responds with the required decryption key.

3.1 Escrow Agents

Escrow agents are responsible for managing keys and provide the data subjects with means to: a) set the authorization rules regarding the usage of their data, and b) monitor if they are enforced.

3.1.1 Options

Data subjects can choose either an external escrow agent, or the default escrow agent that the data controller provides. In the former case, they should register in advance on the escrow agent's website. Then, the first time that the data subjects provide personal data to the data controller, they should also provide the domain name of the external escrow agent of their choice. After that, they can login to the escrow agent's website and set all the permissions regarding their data, in a way we describe in Section 3.1.2. Until then, the escrow agent will deny any access to their personal data. In our reference scenario (Section 2.3), we assume that Alice has chosen an external escrow agent, which means that she has gone through all the aforementioned steps. Data subjects not interested to control and monitor their data are assigned an escrow agent run by the data controller, which implements the organization's personal data security policy.

A data controller may have some obligatory rules concerning a data subject (e.g. a bank offering a mortgage may require non-revocable access to the data subject's address). These rules are provided to the escrow agent the first time that the data controller interacts with a data subject, are marked as such in the escrow's user interface, and cannot be changed thereafter.

3.1.2 Setting Authorization Rules

Data subjects can set their own authorization rules regarding the usage of their personal data through the escrow agent they choose. They can also update or revoke previously defined rules. An authorization rule, which can be set through an escrow agent, involves the following elements.

1. *Data Type* defines the type of a specific piece of personal data, such as given name, surname, address, credit card number. Data types are organized in a hierarchy which is illustrated in Figure 2. This design choice offers the possibility of backward compatible additions. Also, it allows users to set one guideline for multiple types of data via grouping. Notably, sensitive data

(e.g. data related to public health and social discrimination [8]) are distinguished as a different category because in many cases additional regulatory restrictions apply to the processing of such data [8].
2. *Data Use* indicates how the application intends to use the corresponding piece of data. Examples include: "send email to subject", "interact with the subject over the phone", "use data for analytics", "broadcast data" and more. A user can allow multiple data uses for one data type.
3. *Permission to Update and Data Provenance* is defined by the data subject to indicate whether and how a piece of data can be updated. If the permission is granted, the data subject must identify the possible sources that may update the corresponding data; for example the data can be obtained from a public registry.
4. *Authenticated Parties* include all data controllers that an external escrow agent allows to process personal data.

Given the preceding concepts, data subjects can specify rules regarding the actions (including data uses and updates) that can be performed by data controllers, on their data types. In essence, PDGuard is based on the Discretionary Access Control (DAC) model, which allows users to control access to their own data [63, 9]. The access control matrix regarding the personal data of a specific data subject can be abstracted in the following manner: If T is the set of data types of this subject, C is the set of data controllers affiliated with this data subject, and A is the set of all possible actions, every entry of the matrix is a set of actions of the form $a(t,\ c)$, where $t \in \text{T}$, $c \in \text{C}$, and $a(t, c) \subseteq \text{A}$. Note that, if a data subject does not set a rule for a data type, then it is treated as non-accessible.

Furthermore, data subjects can provide a *time interval* for every guideline. This feature, determines the validity period of either an allowable data use or update from a specific source. For instance, a data subject can specify that data controller applications may "send email to subject" for one year.

In our reference scenario (see Section 2.3), Alice should set the following rules: (1) allow the website to send her emails, (2) access her credit card with a time interval of two years, and (3) allow her name to be displayed on the forum's pages. Also, she should allow the profile service to update her address. By not setting any rules for the rest of the data she implies that none of the website's other applications can process them in any way.

3.1.3 Monitoring Service

Escrow agents provide a service that can be used by data subjects who wish to monitor all the actions related to their personal data. A data subject can view the applications that requested to use their data (and the corresponding data controller), the date and the time of the request, the data type, the intended data use, and the *interaction purpose*. An interaction purpose is another feature that is currently supported by our framework for logging purposes. In case of an update, or in the case where data were provided by another entity (e.g. the data were obtained from a public registry), the data subject can review

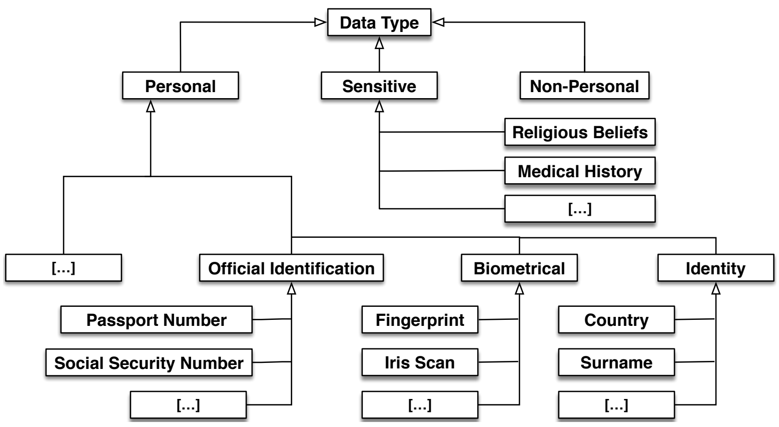


Fig. 2: **Data Type Hierarchy**.

the sources that performed these actions. Finally, data subjects can also check if the agent granted the authorization or not.

Through this monitoring service, data subjects can recognize a potential misuse of their data. Consider the case, where Alice sees that the news website where she is registered, has requested the decryption of her street address 500 times within one day. This would imply that the application is trying to use this piece of data in an inappropriate manner.

3.1.4 Key Management

Escrow agents maintain key servers where the cryptographic keys are stored encrypted just like the personal data in the data controller. The master key used to decrypt the keys is also stored on the escrow agent's side in a secure location.

An escrow agent assigns a different secret key (K) to every (DS, DC) pair, where DS is a data subject and DC is the data controller that maintains and processes the personal data of DS. For example, one key is generated and used when the news and media website processes the personal data of Alice, while a different key will be employed when Alice signs up with a government agency or e-shop.

K is not the key used to encrypt or decrypt personal data. Instead, to reduce the attack surface of the data processing system even more, we use a hash function with K and the data type (e.g. surname) of the requested piece of data (PoD) as arguments. The function yields K', which is the key that will be used for either the encryption or the decryption of this PoD. Thus, a different key is used for every data type of a specified data subject. This feature is fundamental in terms of security, as we explain in the "Security Analysis" Section (5.4).

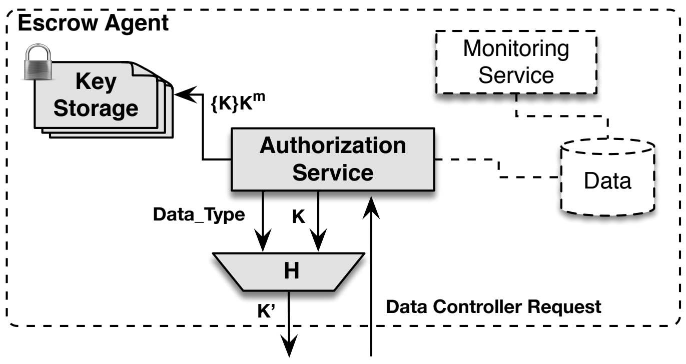


Fig. 3: **Key Management on the Escrow Agent**. When there is a rule that allows a requested data use or update, the authorization service retrieves the key (also encrypted by a master key — {K}K^{m}) that corresponds to the (DS, DC) pair from the key storage. It decrypts it and feeds it to a hash function (H) together with the type of the requested data. The result is the key (K') that will be used for either the encryption or the decryption of this piece of data. Notably, K never reaches the data controller.

3.2 The PDGuard API

There are two basic API calls that a data controller (DC) application should invoke to process the personal data of a specific data subject (DS), namely: `decryptData` and `encryptData`. Both, are members of the `DataProtection` class, which is used to associate an escrow agent with a number of authenticated entities that include the data subject, the data controller, the data controller application, and more. In this way, the `DataProtection` class provides the necessary background for each call.

To access an opaque encrypted object (OEO) developers should invoke the `decryptData` method after the database action used to retrieve the OEO, with the following arguments: the OEO, the object's data type, the intended data use, and the interaction purpose. In case of a successful authorization, the escrow agent returns the required key (K'), and the application uses it accordingly.

In the case where the application needs to store or update a PoD, the `encryptData` call should be invoked right before the database action (e.g. insert, update). In this case developers need to provide the object's data type the PoD to encrypt, a boolean value that indicates if the action is an update or not, and the provenance of this PoD. If K' is retrieved successfully, the API encrypts the PoD with K' and the encrypted value {PoD}K' is used in e.g. a database's data manipulation statement.

For every invocation, the API uses all the provided arguments to form an *authorization bundle*. This bundle is then transferred to the escrow agent in a way that we describe in the following subsection. Depending on the request, the agent will use the various elements of the bundle to check if the request

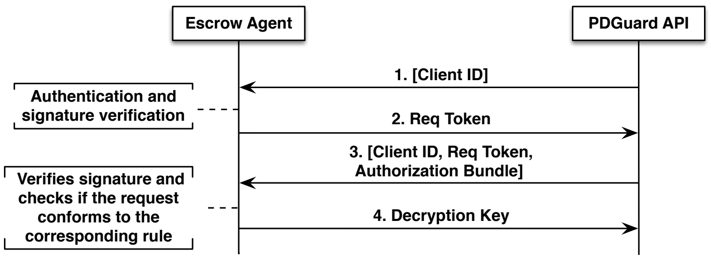

Escrow Agent
PDGuard API
1. [Client ID]
Authentication and signature verification
2. Req Token
3. [Client ID, Req Token, Authorization Bundle]
Verifies signature and checks if the request conforms to the corresponding rule
4. Decryption Key

Fig. 4: **Authorization Protocol Flow**.

conforms with the rules set by the data subject. Specifically, in case of a decrypt operation, the escrow agent will check if the specific data use is allowed for this data type. If an `encryptData` call takes place, the escrow agent will check whether this data type can be updated from this source or not. In both calls, the agent must also check if the rule is valid for the time being. Note that, if there is no corresponding data type, this implies that this is the first time where data of this kind are about to be stored on the controller's side. In this case, the data type and provenance are recorded, but the agent will not allow any access to this object until the data subject explicitly sets rules for it.

3.3 Authorization

To acquire the key needed for the processing of personal data, applications must invoke the PDGuard API. Then, the API takes over and communicates with the escrow agent over a secure channel (TLS), on behalf of the application. Every token and resource request must be signed by the party that performs the request using the secret obtained during a registration stage. The protocol flow is illustrated in Figure 4. All lines, represent direct server-to-server API calls that cannot be tampered with by a third party, and the parameters inside square brackets are signed using shared secrets (in the future these will use a public key infrastructure.)

The first time that a data controller application requests to process personal data referring to a specific data subject, it obtains from the escrow agent a *client* ID, and a *secret*. A client ID is a unique identifier for every triplet that includes: the data subject, the data controller, and the data controller application. The client ID and the secret form the *client credentials* which in turn, will be used for all future communications regarding this data subject.

Consider the case where the news and media website mentioned in our scenario (Section 2.3), attempts to access Alice's email. To do so, the application will use the `decryptData` call. When the call is invoked, the API obtains a request token from the escrow agent on behalf of the application (Steps 1 and 2). The token is signed and it is sent back to the escrow agent together with the authorization bundle (Step 3). At this point, the escrow agent's checks the bundle against the predefined rules provided to the escrow agent by Alice (see Section 3.1.2). If the request conforms to the corresponding rule, the agent sends the key back to the API (Step 4). Recall that, Alice has allowed the website to access her email, therefore the authorization is successful. A request token can be used for one or several key exchanges according to the implementation (e.g. a token can be valid for a specific time interval).

Note that even though our token-based protocol shares a number of features with OAuth [3, 18] , it is different from other access management protocols such as User-Managed Access (UMA) [7] and OpenID Connect (OIDC) [4] – which is built on top of OAuth. For instance, both protocols are designed to interact with end users (recall that our protocol involves only the PDGuard API and the escrow agent). Furthermore, UMA does not provide means to define different access policies.

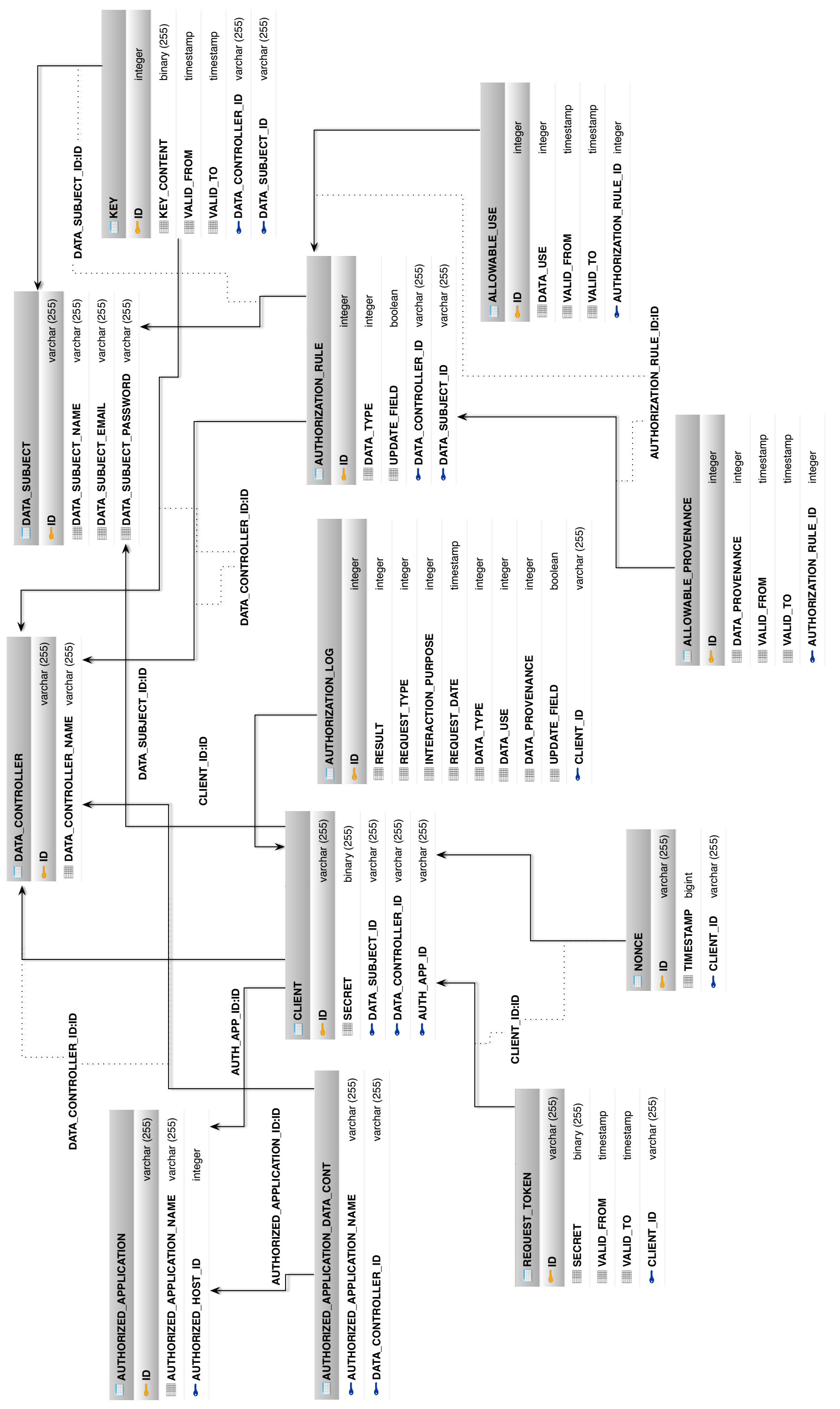


Fig. 5: **Escrow Agent Database Schema**. Foreign keys are shown with a (blue) key icon.

Algorithm 1 Exploring a Method

```
function EXPLORE($m$, $M$, $W$, $V$, $R$)
    $S \leftarrow m.getStatements()$;
    for all $s \in S$ do
        if $s.hasMethodInvocation()$ then
            $m' \leftarrow s.getMethod()$;
            $A \leftarrow m'.getArguments()$;
            $V' \leftarrow A \cap V$;
            if $m' \in M$ then
                EXPLORE(m', $M$, $W$, $V'$, $R$);
                $remove(M, m')$;
            else if $m' \notin W$ and $V' \neq \emptyset$ then
                $verifyUse(V')$;
            end if
        end if
        if $s$ is $AssignmentStatement$ then
            $trackPDGuardVar(s, V, R)$;
        else if $s$ is $ReturnStatement$ then
            $checkReturnedData(m, s, V, R)$;
        end if
    end for
end function
```

3.4 Static Verification of the API Usage

We have developed a verification approach to detect accidental misuse of personal data in data controller applications using the framework. As a misuse you can consider the following: a developer intends to use a decrypted email value as an argument to a network-related method that sends an email to a data subject. However, the value ends up also as an argument to a method that writes it to an external file used for analytical processing. To detect cases like the above, our approach employs code annotations, taint tracking [13, 58], and inter– and intra-procedural analysis. Both developers and external auditors can use the approach to check that applications to not accidentally violate PDGuard use guidelines.

Our verification approach expects that when developers use a method that outputs a decrypted value to an external channel (e.g. a file, the network etc.), an *annotation* will exist be placed before the corresponding code statement. This annotation will contain the intended data use of the value. This data use must be the same as the one stated in the `decryptData` invocation where the value was obtained. If an annotation is missing, or the data use is not the initially declared one, the verifier issues a warning.

The verifier takes two sets as arguments: a set containing all the methods of the application (M), and a set with methods that, by design will not export data (W) e.g. `length`, `hashCode`, and can handle them directly. Then it processes each method m by passing it as an argument to a function we call EXPLORE. EXPLORE examines all the statements of a method as described in Algorithm 1. Notice that the function has also two empty sets as arguments: V and R. As EXPLORE progresses, both sets are populated. In R all the application methods that return variables with decrypted personal data are added (with accompanying data use information). V is populated with all the variables of m, that are derived from a `decryptData` call (i.e. contain personal data) either explicitly or implicitly. Note that for every variable in V, we also keep the corresponding data use. This is done by `trackPDGuardVar` (Algorithm 1, line 16) which is called if the statement s is an assignment. Then, `trackPDGuardVar` inspects if a `decryptData` call or any method $r \in R$ is invoked in the right hand side (RHS) of the statement. In this case, it updates set V by adding the variable found in the left hand side (LHS). In any other case, `trackPDGuardVar` gathers all the variables coming from the statement's RHS and checks if any of them exist in V. If they do, again, it adds the variable found in the LHS into V. In this way we also perform taint propagation, e.g. `String foo = "I want this " + email;`, where the variable `email` holds the decrypted email of a data subject (note that here `foo` inherits the data use of `email`).

A statement may involve a method invocation. In this case, EXPLORE identifies the name of the method (m') and gets every variable $v \in V$ passed as an argument to this method (Algorithm 1, line 7). If m' is an application method, then it must also be explored. Note that V will not be an argument of EXPLORE, because the scope of m' is different. However, all tracked variables that were passed as arguments to m', are propagated by passing set V' into EXPLORE (Algorithm 1, line 9). Then, m' is removed from M as we do not need to explore it again.

If m' is neither an application method nor included in W, then EXPLORE checks the annotation preceding the statement (Algorithm 1, line 12). The data use contained in the annotation should match with the usage of the variables found in V'. In case of a mismatch (or a missing annotation), a warning is issued. Finally, s can be a return statement. If it is, `checkReturnedData` examines if m returns a variable that exists in V and, if this is the case, it adds m to R.

4 Reference Implementation and Use

To demonstrate the feasibility of the approach and to bootstrap its adoption, we have implemented the basic components of PDGuard, namely an escrow agent prototype, a library implementing the PDGuard API library, the protocol that allows the two to interact, and a verifier prototype. All components were developed in the Java programming language.

4.1 Escrow Agent Prototype

The PDGuard escrow agent prototype provides an easy to use web interface for the data subjects to sign up, define or edit rules, and view authorization logs. Data subjects can set one rule for multiple types of data via grouping; for example restrict the handling of all medical data. This is feasible, because data types are organized in a hierarchy, which in turn is implemented via subsumption. We further describe this interface in our Appendix A.

Figure 5 illustrates the database schema used by the PDGuard prototype. Keys are stored in the `KEY` table and their generation is based on the Advanced Encryption Standard (AES) [32]. Observe that a key corresponds to a data controller / data subject couple, and an authorization rule for one data type may involve zero or multiple allowable uses and provenances. Thus, if an application requests to update a specific data type (found on the `AUTHORIZATION_RULE` table), the agent will check if there is a corresponding record on the `ALLOWABLE_PROVENANCE` table. On the other hand, if the application invokes a `decryptData` call, the agent will look for a tuple in the `ALLOWABLE_USE` table. Recall that, the keys found in the `KEY` table are not used either for the encryption or the decryption of a PoD. Instead, they are used as arguments in a hash function (SHA-256 in our implementation) together with the data type of the requested PoD (see Subsection 3.1.4). Finally, request tokens (`REQUEST_TOKEN`) and nonces (`NONCE`) are associated with the client credentials (`CLIENT`), and are used as described in Section 4.3.

4.2 Application-Side API Implementation

Using the PDGuard library is straightforward. As described in Subsection 3.2, `decryptData` and `encryptData` are methods of the `DataProtection` class. To initialize a corresponding object, the developer must pass the domain name of the escrow agent and the client credentials as arguments to the constructor of the class. In turn, the client credentials can be initialized via a class named `RegistrationService`.

Consider the case where Alice provides her surname to the news and media website when she signs up for the first time (recall our reference scenario in Section 2.3). The code running on the background should first initialize a `DataProtection` object (e.g. `dp`) and then invoke the `encryptData` method in the following manner (assume that `surname` is a string variable which has Alice's surname as its value):

```
dp.encryptData(surname, DataType.SURNAME, DataProvenance.DATA_SUBJECT_EXPLICIT,
      false));
```

Listing 1: `encryptData` invocation example.

Observe that, the data provenance (`DATA_SUBJECT_EXPLICIT`) indicates that this piece of data comes from the data subject itself. The final argument informs the escrow agent that this action is not an update.

Now assume that the website needs to send a weekly digest via email to Alice, per her request. To do so, it needs to retrieve her email, which is stored in the database as an OEO. In this case, the `decryptData` method should be invoked in the following manner:

```
String email = new String(dp.decryptData(OEO, DataType.PERSONAL_EMAIL, DataUse.
     COMPOSE_EMAIL_TO_SUBJECT, InteractionPurpose.INFORMATIVE));
```

Listing 2: `decryptData` invocation example.

```
1  [Token Request to eagent.org]
2  POST /token HTTP/1.1
3  Host: www.eagent.org
4  client_id=edb924a9-25d5-4e65&nonce=bf1e1a5c-1179-4e73-b9d8&timestamp=
       1447629467784&signature=%2BuWVpmw%2B3kyv4IDrsxk1Jih5qFk%3D
5  [Response from eagent.org]
6  HTTP/1.1 200 OK
7  request_token=5e9e0dee-a932&token_secret=80
       ebd92bb60d2912026726687247e6c994ccf308762ada
8  [Authorization Request to eagent.org]
9  POST /token HTTP/1.1
10 Host: www.eagent.org
11 client_id=edb924a9-25d5-4e65&data_type=PERSONAL_EMAIL&data_use=
       COMPOSE_EMAIL_TO_SUBJECT&interaction_purpose=INFORMATIVE&nonce=3
       c105f42-462c-4500&request_token=5e9e0dee-a932&request_type=DECRYPTION&
       timestamp=1447629467817&signature=fVtl%2BORTjjgrFRQMx4ep5nVwlNA%3D
12 [Redirection]
13 HTTP/1.1 303 See other
14 Location: www.eagent.org/decrypt?client_id=edb924a9-25d5-4e65&data_type
       =PERSONAL_EMAIL&data_use=COMPOSE_EMAIL_TO_SUBJECT&interaction_purpose
       =INFORMATIVE
15 [Response from eagent.org]
16 HTTP/1.1 200 OK
17 decryption_key=3b37d8873cebeec2cde42e49d4260fdf
```

Fig. 6: **Example use of the authorization protocol.** The requests are performed by the API on the background, after the `decryptData` call shown in Listing 2.

4.3 Protocol Implementation

Figure 6, presents a sequence that involves: a) the requests performed by the API, b) escrow agent responses, and c) a redirection to the escrow agent service that checks the authorization bundle. The HMAC [38] signature method is currently used for the verification of each signature.

The application includes in every request the parameter `client_id`: a unique identifier that corresponds to the data subject, data controller, data controller application tripplet (see Section 3.3). Recall that, this triplet is sent to the escrow agent when the application first attempts to process a data subject's personal data. During this interaction, the escrow agent will send back this string together with the secret to be used for every future interaction concerning the specified data subject. Other parameters that can be found in each request include a timestamp and a nonce.

The authorization request seen in line 11 of Figure 6, is actually the request performed after the `decryptData` invocation discussed in Subsection 4.2. In line 14 we can see a redirection to the authorization service that our current implementation uses to check the bundle. The service retrieves it and checks the rules set by the subject. In this case, the authorization is granted.

4.4 Static Verifier Implementation

We have developed a prototype of the verification approach described in Subsection 3.4. Given a Java application that employs PDGuard, we can process it by using a well-established Java parser [59], and check for accidental misuse of personal data.

For our current implementation, developers have to annotate code statements that invoke methods channeling decrypted data outside the scope of the application (recall section 3.4). Typical examples include (1) setter methods (e.g. `setString`), the `execute` method along with its variations (coming from either the `java.sql.PreparedStatement` class or the `java.sql.Statement` class), (2) `write` and its variations (e.g. `writeBytes`) derived from classes of the `java.io` package, and (3) any method that comes from a third party package.

Annotations are expressed as JavaDoc comments before each method invocation. For example, if the developer intends to write the decrypted value contained in the `email` variable to a file, to send an email to data subject (`out.write(email);`) the following annotation should precede the statement: `// @pdguse(email, COMPOSE_EMAIL_TO_SUBJECT).`

Our whitelist (i.e. set W) consists of methods that do not expose data outside the application's scope such as `toString`, `length`, `equals`, `valueOf`, `getBytes` and more. Note that users can add more methods to W via configuration files.

5 Evaluation

To demonstrate the applicability of our framework we have built a PDGuard-based application, and also integrated it with a popular web application. In addition, we have performed simple experiments to (1) measure the computational overhead that is introduced by a PDGuard API call, and (2) check the effectiveness of our verification mechanism. Finally, we have evaluated the security of PDGuard and its behavior in the presence of a potential attacker.

5.1 PDGuard Applications

Here we describe an e-shop application that utilizes the features PDGuard, and discuss how we introduced PDGuard to *The Guardian* newspaper's web-

Table 1: **Java or Scala code and HTML markup lines (SLoC) of two applications before and after their integration with PDGuard.**

	Original			**w/ PDGuard**			**Percentage**
Application	Code	HTML	Total	Code	HTML	Total	**Increase**
Custom E-shop	1173	624	1797	1382	635	2017	12.2%
Guardian's "Identity"	3949	1966	5915	4225	1979	6204	4.8%

```
encryptedData = getDataToDecrypt(fields);
decryptedData = Map();
for fieldName, encryptedValue in encryptedData {
    decryptedData.add(fieldName, dp.decryptData(encryptedValue, fields.get(
        fieldName), DataUse.REPORT, InteractionPurpose.INFORMATIVE));
}
updateUserObject(decryptedData);
```

Fig. 7: **Pseudo-code for changes to the "Identity" application to decrypt personal data.** `fields` is a map with the private fields of a user object as keys (the `user` class is defined and used internally by the application), and the corresponding PDGuard data types as values. `getDataToDecrypt` creates a map with the private fields as keys and the corresponding ciphertexts as values.

site identity application.[1] Table 1 presents the Source Lines of Code (SLoC) of the two applications before and after introducing PDGuard. In both cases, seven pieces of personal data were stored encrypted including names, surnames, addresses, and credit cards. The results indicate that little developer effort is required to protect a wide range of confidential data.

5.1.1 Building a PDGuard-based E-Shop

The e-shop application implements two basic functionalities: customer registration / authentication and order processing. In both cases, the application processes personal data.

Our e-shop was written in Java and employs the object-relational mapping (ORM) programming technique which creates a virtual object database that can be used from within the application. The application's database model involves two classes that have personal data as variables, namely: the `Customer` class (email, name, surname) and the `Order` class (city, country, street address, credit card number). Calls to `encryptData` and the subsequent data encryption are performed either when a new customer registers or when an order is processed. Intuitively, the reader can infer that the data provenance of the data in these cases is set to `DATA_SUBJECT_EXPLICIT`, which means that they are provided by the data subject itself. For every authenticated customer, the e-shop has a predefined rule that allows it to access the customer's e-mail for informative purposes (`InteractionPurpose.INFORMATIVE`). To do so, the application performs a `decryptData` call after a successful login.

5.1.2 The Guardian's Identity Application

The news and media website *theguardian.com* is owned by the Guardian Media Group [5]. It is developed in the Scala programming language and all its source code is available through a GitHub repository [6]. From the various sub-projects it contains, we chose to port the "Identity"[2] application. This

[1] `https://profile.theguardian.com/signin`

[2] `https://github.com/guardian/frontend/tree/master/identity`

application is directly related to personal user data because it handles the Guardian profile functionality. In particular, through “Identity”, users can create an account to personalize their experience (e.g. leave comments, enter competitions and subscribe to email services). To do so, they must provide several pieces of personal data, such as emails, home addresses, and birth dates.

We successfully incorporated PDGuard to the “Identity” application by adding 289 lines of code (LOC) to the 5915 existing ones. One modification was made to change the sign up webpage, to let users provide the domain name of an escrow agent (13 HTML LOC). The rest, involved the encryption and decryption of the various personal data obtained from the user. For instance, when users sign up, they fill in a form with all their data. Then a class named `RegistrationController`[3] retrieves the data and initializes a `user` object, which in turn, is passed to a class named `IdAPI`.[4] Finally, `IdAPI` interacts with the Guardian's back-end and stores all the provided information. We introduced a number of `encryptData` calls in this class right before the database interaction (the data provenance is set to `DATA_SUBJECT_EXPLICIT` here too).

Consider the case where the application attempts to access personal data and display them as part of a webpage (as is done in the `EditProfileController`[5] class). After the application retrieves the data from the database, our glue code gathers them and decrypts them. as shown in Figure 7. Now, assume that a data subject chose to set her surname as inaccessible by the website. As a result, her surname will not appear on the corresponding webpage as seen in Figure 8.

5.2 Performance

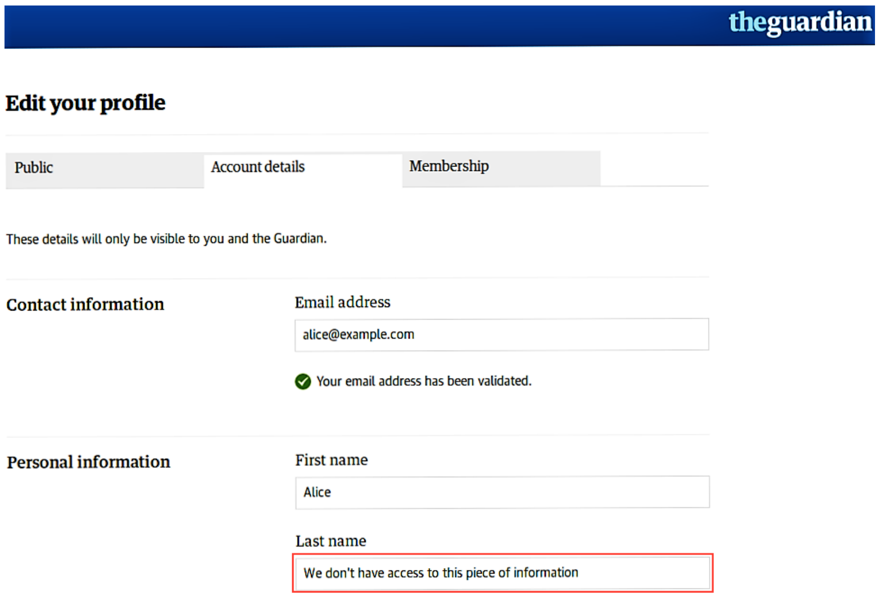
To measure the overhead added by PDGuard API calls, we instrumented the “Identity” application with high precision time counters. All measurements were run on a client with a quad core processor with 8GB of RAM, running Ubuntu Linux. The escrow agent was running on a remote server with a dual core processor with 2GB of RAM, running Debian Linux.

In our experiments, we measured how much time the application spends to either access or update a piece of data, with and without PDGuard. To get reliable measurements, we issued 300 consecutive API invocations for each case (i.e., access and update requests). On average, it took the application 451 *ms* to access, and 655 *ms* to update a piece of data without PDGuard. Figure 9, illustrates how much time the application spends to either access or update a piece of data, with and without PDGuard. When PDGuard is used, the corresponding average time to either access or update the same piece of data is 618 and 828 *ms* respectively. As it seems, the overhead is noticeable

[3] app/controllers/RegistrationController.scala

[4] `app/idapiclient/IdApi.scala`

[5] `app/controllers/EditProfileController.scala`



Edit your profile

These details will only be visible to you and the Guardian.

Contact information

Email address: alice@example.com

Your email address has been validated.

Personal information

First name: Alice

Last name: We don't have access to this piece of information

Fig. 8: **Screenshot from the ported "Identity" application**. We observe that Alice does not permit access to her surname.

in both cases: 25% for data access (`encryptData`), and 37% for data update (`decryptData`).

By profiling the client application to examine the reasons behind this overhead, we found that most time is spent to functions that involve the checking of the authorization bundle by the escrow agent. Also, network latency (recall that the escrow agent was running on a remote machine), which was approximately 40 *ms* for both API calls, is included in the overhead. Although under some circumstances this latency can be a usability issue, it does not affect the framework's scalability. Notably, if we exclude this latency from the measurement, the computational overhead of the framework is significantly reduced (28% and 20% for a `decryptData` and an `encryptData` API call respectively).

We need to mention here that, we did not attempt to optimize the framework's code. By providing extra features like caching, the overhead could be significantly reduced. For example, the API could allow the data to be cached by a trusted application for one week. In this way, we could avoid sending a request to the escrow agent, for every action involving personal data.

5.3 Verification

We modified our e-shop application to make it write a decrypted value to a file in different ways and with different annotations (7 in total). Our modifications also involved wrapper methods and implicit assignments. We then used our static code verifier to detect the misuses. The verifier correctly identified the

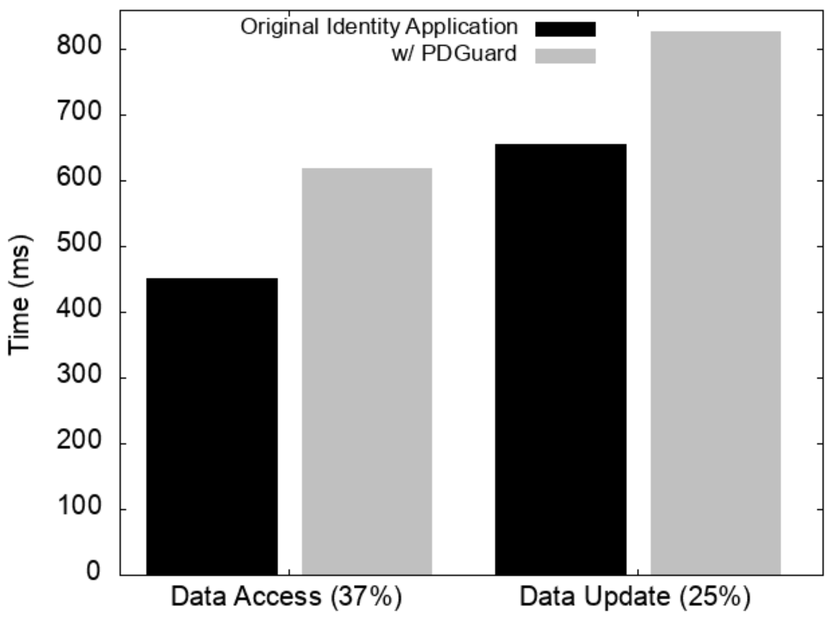


Fig. 9: **The time (in milliseconds) needed to either access (`decryptData`) or update (`encryptData`) data, with and without our framework**. The overhead introduced by each call can be seen below, right next to each action. Note that there is a network latency included in the grey charts (∼40 ms), because of the interaction with the escrow agent (which was running on a remote machine).

cases without any false positives or negatives. However, more extensive testing is required to validate the efficacy of taint propagation.

Consider the following code fragment, where the developer invokes a function that wraps a `decryptData` call (`getCustomerEmail`) and attempts to write the decrypted value to a file in two occasions (lines 4 and 7):

```
1  BufferedWriter out = new BufferedWriter(new FileWriter("file.txt"));
2  String email = getCustomerEmail();
3  // @pdguse(email, COMPOSE_EMAIL_TO_SUBJECT)
4  out.write(email);
5  String bar = email + "bar";
6  // @pdguse(bar, SEND_SMS_TO_SUBJECT)
7  out.write(bar);
```

Given that in the initial `decryptData` call, the data use was indicated as `COMPOSE_EMAIL_TO_SUBJECT`, the statement in line 4 passes verification as legitimate. On the other hand, the verifier raises a warning when it examines line 6.

5.4 Security Analysis

We evaluate the security of PDGuard, by assessing how the framework secures against either external of internal adversaries attempting to access personal data.

External attackers may perform attacks, such as SQL injection [50] and shell-code injection [65]. PDGuard is not designed to protect against such attacks and requires a secure application to work as intended. However, there are cases where PDGuard will prevent such attacks even if an application is vulnerable.

Consider a web application that accepts and processes user input without proper validation and / or filtering. If the application concatenates the input with SQL queries, then it is vulnerable to SQL injection attacks. When an outsider performs such an attack to retrieve the personal data of multiple users (e.g. addresses), the attack will only return useless encrypted data. Nevertheless, PDGuard will not prevent an attack against the integrity of the database (e.g. an attempt to drop a table).

If attackers gain control over the target server via a shell-code injection attack, they may attempt to develop an application that utilizes the PDGuard API. To authorize this application's exchange with the various escrow agents in the way we presented in Section 3.3, a digital certificate is needed. This certificate though, should not be necessarily collocated with the vulnerable application: the application could initially retrieve it from a smart card. Still, attackers could extract it from the process memory [37]. In addition, they may try to retrieve cryptographic keys from an application's memory dump (this could be done by a tech-savvy employee too). In that case however, the attacker will manage to access only one piece of data because the key is related to a specific data type of a specific data subject (recall our design choice to use a hash function—see Section 3.1.4).

More dangerous are native code injection attacks [35]. These allow the attacker to issue an arbitrary number of PDGuard calls and obtain the corresponding data. In theory, such attacks can be thwarted by including checks, that verify the application's integrity [62] in our communication protocol – see Subsection 3.3. This would however require substantial changes to the protocol, and would interfere with instruction randomization techniques.

PDGuard will prevent by design threats such as transferring data between different computers and unauthorized access through application misuse. However, PDGuard will not prevent either a malicious administrator who has access to the server where the application is running, or a rogue developer who will leverage the framework's API for processing personal data (see Section 2.2).

A successful attack against an escrow agent will not lead to any unauthorized action because personal data are never stored there. Even if attackers also manage to compromise a data controller, they will only obtain the subset of data of users using the specific escrow agent. In the case where attackers manage to make the escrow agent unavailable through a denial of service (DoS) attack, all personal data become automatically unavailable until the corresponding server is reachable again (the API will not be able to retrieve any keys from the escrow agent).

Two additional threats to privacy are associated with meta data and aggregate data. First, a privacy breach could occur when an attack targets the meta data stored on the escrow agent's side (e.g. check what kind of data users provide to which sites or when such data is accessed). This can be addressed by encrypting the meta data too. Specifically, a user's auditing data can be encrypted by a key that only the corresponding user holds. Similarly, access permission data can be encrypted by keys that data controllers provide. Extending PDGuard to support this functionality is relatively straightforward. Second, leaked aggregate data held by data controllers (e.g. "30-year-olds favor product X"), can lead to the disclosure of personal data if it is used as a stepping stone to mount database inference [16] or machine learning attacks [26, 57]. However, if data subjects do not consent for the data controller to use their data as a basis to derive aggregate data or train machine learning models, then such threats are avoided.

6 Challenges

In this section, we discuss a number of challenges that emerge in the context of PDGuard and describe how they can be addressed.

Aiding Intrusion Detection PDGuard can help detect and prevent attacks that target personal data. Specifically, the logging service provided by the framework can be utilized by intrusion detection systems (IDS) [20, 42]. For instance, consider a case where an application retrieves credit card data at an unprecedented rate. Such an event could indicate a malicious act, thus a service running on the side of the escrow agent could temporarily deny all requests coming from this application.

Search over Encrypted Data Currently, PDGuard does not allow database-layer searches over encrypted data, e.g. obtaining a list of all patients that have been admitted for a particular symptom. As we mentioned earlier, our e-shop application employs the ORM programming technique and the Identity application uses a NoSQL backend. Consequently, each application performed actions directly on the fields of objects, without issuing SQL queries with arguments containing personal data. To handle this shortcoming, PDGuard could be extended to use schemes such as CryptDB [48], an approach that supports the execution of queries over encrypted data through efficient SQL-aware encryption.

PDGuard API Extention for Bulk Encryption / Decryption The current version of the PDGuard API does not support bulk encryption or decryption of records. Consider the case where a data controller wants to tally the number of its customers by their postal code. In this case the `decryptData` call must be invoked several times adding significant overhead. Extending the API to include an array of authorization bundles as a request's argument, could solve this problem and make the framework more efficient.

7 Related Work

Diverse approaches have been developed to prevent unauthorized personal data processing; many with different goals and threat models than PDGuard. We describe how each one operates, outline its main characteristics, and compare it to PDGuard.

An early case where cryptography and an escrow service were used to protect data was Ephemerizer [47, 44]. In the context of the Ephemerizer, all user data are stored encrypted and the keys are managed centrally by a trusted third party. This entity destroys the cryptographic key after a specified timeout, thus making the data unavailable. Contrary to the Ephemerizer, Vanish [27] follows a decentralized approach to protect the privacy of past, archived data by integrating cryptographic techniques with global-scale, P2P, distributed hash tables (DHTs). DHTs implement an index-value database on a collection of P2P nodes. Vanish encrypts personal data locally with a random encryption key not known to the user. Then, after a user-specified time, it destroys the local copy of the key and sprinkles its bits across random indexes in the DHT. The main goal of these approaches is to make personal data, such as emails and social media messages, (typically data that exist on a personal computer) unavailable after a specific period of time. PDGuard operates in a broader context and involves personal data that are stored by the various data controllers. Through PDGuard, data subjects can render their data unaccessible after a certain period of time by associating specific time intervals with classes of data.

MYLAR [49] is a platform for developing web applications, which protects sensitive data against attackers with full access to servers, thus dealing with more threats than PDGuard. To do so, it stores personal data encrypted on the server, and decrypts them only in users' browsers. In addition, it allows the server to perform searches over encrypted documents. Contrary to PDGuard, MYLAR does not provide a way for data subjects to control and audit the use of their personal data. Also, the fact that data decryption is performed on the client-side, interferes with the running of back-office operations and makes the framework susceptible to XSS attacks.

Cloudfence [46] is a framework tailored to cloud hosting environments that provides data tracking capabilities to both service providers, as well as their users. To track the flow of the data within a cloud infrastructure, Cloudfence employs byte-level data tagging. To enable this functionality, application developers must use specific API calls. In addition, it generates detailed audit trails for tagged data via a logging service. Silverline [43] is a system close to Cloudfence. However, the process-level tainting it supports is rather coarse-grained for the most common web applications. In both cases, data subjects cannot control the use of their personal data. Furthermore, even if both frameworks aid the prevention of external attacks, they will not protect against insider threats. PDGuard achieves the latter by storing personal data encrypted. Song et al. [60] have also proposed a high level architecture for data protection in the cloud. The architecture is based on trusted platform modules [15], but the authors do not provide an implementation and evaluation. In addition, the proposed solution does not provide a way for users to control and audit the use of their data, leaving that to the developers.

Ding et al. [24] have introduced a scheme to support privacy-preserving data processing in the cloud. Specifically, their approach provides means for secure computations over encrypted data in a way that does not violate user-privacy. Such means include re-encryption primitives and signing. Re-encryption primitives [23], including re-encryption proxies [10], have been also developed in the context of the CREDENTIAL [2] project, to provide secure identity management and data-sharing in the cloud. A number of different user-centric identity management approaches have been also developed by the various partners of the ABC4Tust [1] project. In particular, the partners have collaborated to introduce technologies [17, 55] that support Attribute-based Credentials (ABC). Such credentials, allow a holder to reveal minimal personal information to an application. In this way, users do not need to give away their full identity information but specific attributes (e.g. age). User-centric approaches can be complementary to our approach. Indeed, by extending the application on the escrow agent's part, we can provide users with means (e.g. data wallets [34]) to decide about releasing different attributes in different ways.

The Adaptive Trust Negotiation and Access Control (ATNAC) framework [54] aims to prevent the leakage of sensitive data during an electronic transaction. It builds on two existing systems, TrustBuilder [67] and the Generic Authorization and Access-control API (GAA-API) [51, 52, 53]. The former is a trust negotiation system that regulates when and how sensitive information is disclosed to other parties, and the latter is a framework that allows dynamic adaptation to network threat conditions communicated by an IDS. PDGuard has a much broader scope than ATNAC and by design can prevent a potential data leakage during a transaction over the web.

The "Personal Data Storage" (PDS) approach [36] was inspired by the shift of information systems' center of gravity from organizations to individuals (e.g. social networks). A PDS stores data in one central point, with application-specific services plugging into this core. Hence, the data subject is the central point of control in this model. This differs from the service provider-centric control currently in place in many web applications. Multiple PDS can co-exist into a third-party entity, similar to our escrow agent. The main problem in this case is that if attackers manage to hijack this entity, they automatically have access to the personal data of multiple data subjects. In the context of PDGuard though, hacking an escrow agent is useless without a compromised data controller as we have pointed out in Subsection 5.4.

Yu et al. [68] have employed applied cryptography but for a slightly different purpose: to secure user files stored on cloud applications. Specifically, they try to address overhead issues imposed by the distribution of decryption keys by combining different re-encryption schemes with Attribute-Based Encryption (ABE) [28]. ABE is a type of public-key encryption in which the key of a user and the ciphertext are dependent upon attributes (e.g. country, gender). In this context, the decryption of a ciphertext is possible only if the set of attributes of the user key matches the attributes of the ciphertext. In PDGuard key management differs in the following manner: data types are used as attributes to generate symmetric keys for the encryption or decrytpion of a specific piece of data. EASiER is also based on ABE to support fine-grained access control policies and dynamic group membership in the context of a social media network (i.e. a user defines which data can be viewed by other friends and groups that are registered in the social network). In this context the threat model is different than PDGuard's. Our approach is more generic but it can be applied to this context too.

8 Concluding Remarks

IT security in general and the protection of personal data in particular are tough problems to crack, for they involve diverse stakeholders with conflicting interests, determined unknown opponents, and an unclear balance of risks versus payoffs. As these problems involve a multitude of complex systems, technology cannot offer a silver bullet, but it can help make them tractable. Adopting a security by design approach, PDGuard provides through a standardized, auditable mechanism an effective way for protecting personal data by offering to data subjects direct control and monitoring, and by reducing and strengthening the attack surface of data management systems maintained by data controllers. To do so, PDGuard combines different concepts including applied cryptography, access control models, an API, authorization protocols, and taint tracking. Our evaluation demonstrates that PDGuard can indeed be used in practice both in new and in legacy applications.

Can PDGuard actually make a difference in our everyday lives? This goal requires a lot more work ranging from scientific and technical to evangelism. Important elements include: the design of performance optimizations at the architectural level; the development of additional auditing tools and methods; the running of large scale trials; the operation of an escrow agent in a production setting; the implementation of PDGuard API libraries in more commonly used programming languages; the initial adoption by significant data controllers; the establishment of a community and a governance structure; as well as the education of developers, data controllers, and data subjects. Achieving these objectives seems like a tall order, but this is the way in which technology changes our lives.

9 Compliance with Ethical Standards

– **Funding**: This work has received funding from the EU's Horizon 2020 research and innovation programme under grant agreement No 825328 and the Research Centre of the Athens University of Economics and Business, under the Original Scientific Publications framework 2019 (project ER-3074-01).
– **Conflict of Interest**: The authors declare that they have no conflict of interest.
– **Ethical approval**: This article does not contain any studies with human participants or animals performed by any of the authors.

Availability. The source code of our framework is available as open-source software at `https://github.com/AUEB-BALab/PDGuard`.

Acknowledgments: We would like to thank Amit Levy, Panos Louridas, Theofilos Petsios, and George Argyros for their insightful comments.

References

1. ABC4Trust EU project: Official website. `https://www.abc4trust.eu/index.php`. [Online; accessed 09-July-2019].
2. CREDENTIAL: Secure cloud identity wallet. `https://credential.eu/`. [Online; accessed 09-July-2019].
3. OAuth: An open protocol to allow secure authorization in a simple and standard method from web, mobile and desktop applications. `http://oauth.net/`. [Online; accessed 09-July-2019].
4. OpenID connect main website. `https://openid.net/connect/`. [Online; accessed 09-July-2019].
5. The Guardian Media Group. `http://www.theguardian.com/gmg`. [Online; accessed 30-September-2018].
6. The Guardian. The source code of the world's leading liberal voice. `https://github.com/guardian`. [Online; accessed 30-September-2018].
7. User managed access: Created by kantara initiative staff. `https://kantarainitiative.org/confluence/display/LC/User+Managed+Access`. [Online; accessed 05-July-2019].
8. The European Union General Data Protection Regulation (GDPR). `http://data.consilium.europa.eu/doc/document/ST-5419-2016-INIT/en/pdf`, 2016. [Online; accessed 30-September-2018].
9. Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 2001.
10. Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger. Improved proxy re-encryption schemes with applications to secure distributed storage. *ACM Trans. Inf. Syst. Secur.*, 9(1):1–30, February 2006.
11. Paul Barford, Igor Canadi, Darja Krushevskaja, Qiang Ma, and S. Muthukrishnan. Adscape: Harvesting and analyzing online display ads. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 597–608, New York, NY, USA, 2014. ACM.
12. Sean Barnum and Michael Gegick. Design principles. `https://buildsecurityin.us-cert.gov/articles/knowledge/principles/design-principles`, September 19, 2005.
13. Jonathan Bell and Gail Kaiser. Phosphor: Illuminating dynamic data flow in commodity jvms. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 83–101, New York, NY, USA, 2014. ACM.

14. Steven M. Bellovin. *Thinking Security: Stopping Next Year's Hackers*. Addison-Wesley, Boston, 2016.
15. Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vtpm: Virtualizing the trusted platform module. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.
16. Alexander Brodsky, Csilla Farkas, and Sushil Jajodia. Secure databases: Constraints, inference channels, and monitoring disclosures. *IEEE Transactions on Knowledge and Data Engineering*, 12, 12 2000.
17. Jan Camenisch, Anja Lehmann, Gregory Neven, and Alfredo Rial. Privacy-preserving auditing for attribute-based credentials. In *19th European Symposium on Research in Computer Security - Volume 8713*, ESORICS 2014, pages 109–127, New York, NY, USA, 2014. Springer-Verlag New York, Inc.
18. Eric Y. Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. OAuth demystified for mobile application developers. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 892–903, New York, NY, USA, 2014. ACM.
19. Frederick B. Cohen. Defense-in-depth against computer viruses. *Comput. Secur.*, 11(6):563–579, October 1992.
20. Dorothy Elizabeth Robling Denning. An intrusion detection model. *IEEE Transactions on Software Engineering*, 13(2):222–232, February 1987.
21. Peter J. Denning. *Computers Under Attack: Intruders, Worms, and Viruses*. Addison-Wesley, 1990.
22. United States Department Of Veterans Affairs. Management of data breaches involving sensitive personal information (SPI). `http://www.va.gov/vapubs/viewPublication.asp?Pub_ID=608`, January 6, 2012.
23. David Derler, Stephan Krenn, Thomas Lorünser, Sebastian Ramacher, Daniel Slamanig, and Christoph Striecks. Revisiting proxy re-encryption: Forward secrecy, improved security, and applications. Cryptology ePrint Archive, Report 2018/321, 2018. `https://eprint.iacr.org/2018/321`.
24. W. Ding, Z. Yan, and R. Deng. Privacy-preserving data processing with flexible access control. *IEEE Transactions on Dependable and Secure Computing*, December 2017.
25. Nishant Doshi. Facebook applications accidentally leaking access to third parties. Technical report, Symantec Corporation, 2011. [Online; accessed 10-February-2017].
26. Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1322–1333, New York, NY, USA, 2015. ACM.
27. Roxana Geambasu, Tadayoshi Kohno, Amit A. Levy, and Henry M. Levy. Vanish: Increasing data privacy with self-destructing data. In *Proceedings of the 18th Conference on USENIX Security Symposium*, pages 299–316,

Berkeley, CA, USA, 2009. USENIX Association.

28. Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 89–98, New York, NY, USA, 2006. ACM.
29. Samuel Grogan and Aleecia M. McDonald. Access denied! contrasting data access in the United States and Ireland. In *Proceedings on Privacy Enhancing Technologies*, pages 191–211. De Gruyter, 2016.
30. Aniko Hannak, Gary Soeller, David Lazer, Alan Mislove, and Christo Wilson. Measuring price discrimination and steering on e-commerce web sites. In *Proceedings of the 2014 Internet Measurement Conference*, pages 305–318, New York, NY, USA, 2014. ACM.
31. Michael Howard and David LeBlanc. *Writing Secure Code.* Microsoft Press, Redmond, WA, second edition, 2003.
32. International Organization for Standardization. *Information technology — Security techniques — Encryption algorithms — Part 3: Block ciphers.* ISO, Geneva, Switzerland, 2010. ISO/IEC 18033-3:2010.
33. Poul-Henning Kamp. Linkedin password leak: Salt their hide. *Queue*, 10(6):20:20–20:22, June 2012.
34. Farzaneh Karegar, Daniel Lindegren, John Sören Pettersson, and Simone Fischer-Hübner. Assessments of a cloud-based data wallet for personal identity management. In *Information Systems Development: Advances in Methods, Tools and Management - Proceedings of the 26th International Conference on Information Systems Development, ISD 2017, Larnaca, Cyprus, University of Central Lancashire Cyprus, September 6-8, 2017*, 2017.
35. Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 272–280, New York, NY, USA, 2003. ACM.
36. Tom Kirkham, Sandra Winfield, Serge Ravet, and Sampo Kellomaki. The personal data store approach to personal data security. *IEEE Security and Privacy*, 11(5):12–19, September 2013.
37. Tobias Klein. All your private keys are belong to us. `http://trapkit.de/research/sslkeyfinder/keyfinder_v1.0_20060205.pdf`, Ferbruary 5, 2006.
38. H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. `http://www.ietf.org/rfc/rfc2104.txt`; accessed 09-November-2015, February 1997. RFC 2104 (Informational).
39. Mathias Lécuyer, Riley Spahn, Yannis Spiliopolous, Augustin Chaintreau, Roxana Geambasu, and Daniel Hsu. Sunlight: Fine-grained targeting detection at scale with statistical confidence. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 554–566, 2015.
40. Michelle L. Mazurek, Saranga Komanduri, Timothy Vidas, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Patrick Gage Kelley, Richard Shay,

and Blase Ur. Measuring password guessability for an entire university. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 173–186, New York, NY, USA, 2013. ACM.

41. Gary McGraw. *Software Security: Building Security In.* Addison-Wesley Professional, 2006.
42. Aleksandar Milenkoski, Marco Vieira, Samuel Kounev, Alberto Avritzer, and Bryan D. Payne. Evaluating computer intrusion detection systems: A survey of common practices. *ACM Comput. Surv.*, 48(1):12:1–12:41, September 2015.
43. Yogesh Mundada, Anirudh Ramachandran, and Nick Feamster. Silverline: Data and network isolation for cloud services. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'11, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association.
44. Srijith Krishnan Nair, Muhammad Torabi Dashti, Bruno Crispo, and Andrew S. Tanenbaum. A hybrid PKI-IBC based ephemerizer system. In *Proceedings of the IFIP TC-11 22nd International Information Security Conference, 14-16 May 2007, Sandton, South Africa*, pages 241–252, 2007.
45. Arvind Narayanan and Vitaly Shmatikov. Myths and fallacies of "personally identifiable information". *Commun. ACM*, 53(6):24–26, June 2010.
46. Vasilis Pappas, Vasileios P. Kemerlis, Angeliki Zavou, Michalis Polychronakis, and Angelos D. Keromytis. Cloudfence: Data flow tracking as a cloud service. In *Research in Attacks, Intrusions, and Defenses - 16th International Symposium, Rodney Bay, St. Lucia, October 23-25, 2013. Proceedings*, pages 411–431, 2013.
47. Radia Perlman and Radia Perlman. The Ephemerizer: Making data disappear. *Journal of Information System Security*, 1:51–68, 2005.
48. Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100, New York, NY, USA, 2011. ACM.
49. Raluca Ada Popa, Emily Stark, Jonas Helfer, Steven Valdez, Nickolai Zeldovich, M. Frans Kaashoek, and Hari Balakrishnan. Building web applications on top of encrypted data using mylar. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 157–172, Berkeley, CA, USA, 2014. USENIX Association.
50. Donald Ray and Jay Ligatti. Defining code-injection attacks. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190, New York, NY, USA, 2012. ACM.
51. T. Ryutov and C. Neuman. The specification and enforcement of advanced security policies. In *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks*, pages 128–, Washington, DC, USA, 2002. IEEE Computer Society.
52. Tatyana Ryutov, Clifford Neuman, and Dongho Kim. Dynamic authorization and intrusion response in distributed systems. In *DARPA Informa-*

tion Survivability Conference and Exposition, 2003. Proceedings, volume 1, pages 50–61. IEEE, April 2003.

53. Tatyana Ryutov, Clifford Neuman, Dongho Kim, and Li Zhou. Integrated access control and intrusion detection for web servers. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 394–, Washington, DC, USA, 2003. IEEE Computer Society.
54. Tatyana Ryutov, Li Zhou, Clifford Neuman, Travis Leithead, and Kent E. Seamons. Adaptive trust negotiation and access control. In *Proceedings of the Tenth ACM Symposium on Access Control Models and Technologies*, pages 139–146, New York, NY, USA, 2005. ACM.
55. Ahmad Sabouri and Kai Rannenberg. ABC4Trust: Protecting privacy in identity management by bringing privacy-abcs into real-life. In *Privacy and Identity Management for the Future Internet in the Age of Globalisation - 9th IFIP WG 9.2, 9.5, 9.6/11.7, 11.4, 11.6/SIG 9.2.2 International Summer School, Patras, Greece, September 7-12, 2014, Revised Selected Papers*, pages 3–16, 2014.
56. Bruce Schneier. *Secrets & Lies: Digital Security in a Networked World.* Wiley, New York, 2000.
57. Reza Shokri, Marco Stronati, and Vitaly Shmatikov. Membership inference attacks against machine learning models. *2017 IEEE Symposium on Security and Privacy (SP)*, pages 3–18, 2017.
58. Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D. Breaux, and Jianwei Niu. Toward a framework for detecting privacy policy violations in Android application code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 25–36, New York, NY, USA, 2016. ACM.
59. Nicholas Smith, Danny Van Bruggen, and Federico Tomassetti. *JavaParser: Visited.* Leanpub, 2017.
60. Dawn Song, Elaine Shi, Ian Fischer, and Umesh Shankar. Cloud data protection for the masses. *Computer*, 45(1):39–45, January 2012.
61. Sarah Spiekermann. The challenges of privacy by design. *Commun. ACM*, 55(7):38–40, July 2012.
62. Diomidis Spinellis. Reflection as a mechanism for software integrity verification. *ACM Transactions on Information and System Security*, 3(1):51–62, February 2000.
63. Salvatore Stolfo, Steven M. Bellovin, Angelos D. Keromytis, Sara Sinclair, Sean W. Smith, and Shlomo Hershkop. *Insider Attack and Cyber Security: Beyond the Hacker (Advances in Information Security).* Springer-Verlag TELOS, Santa Clara, CA, USA, 1 edition, 2008.
64. Martin R. Stytz. Considering defense in depth for software applications. *IEEE Security and Privacy*, 2(1):72–75, January 2004.
65. Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–382, New York, NY, USA, 2006. ACM.

66. John Viega and Gary McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way.* Addison-Wesley, Boston, MA, 2001.
67. Marianne Winslett, Adam Lee, Lars Olson, and Michael Rosulek. Trust-Builder: negotiating trust in dynamic coalitions. In *DARPA Information Survivability Conference and Exposition, 2003. Proceedings*, volume 2, pages 49–51. IEEE, April 2003.
68. Shucheng Yu, Cong Wang, Kui Ren, and Wenjing Lou. Achieving secure, scalable, and fine-grained data access control in cloud computing. In *Proceedings of the 29th Conference on Information Communications*, INFOCOM'10, pages 534–542, Piscataway, NJ, USA, 2010. IEEE Press.

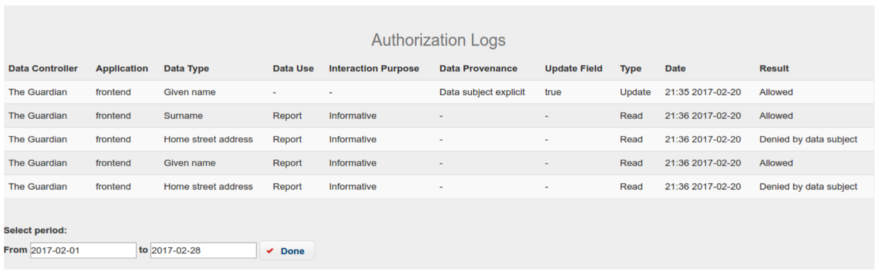
A Appendix

In the following, we describe the user interface provided by our escrow agent reference implementation. Through this interface, data subjects can set or edit authorization rules, and monitor the actions performed on their data.

Data subjects can easily render their data inaccessible or set new allowable actions. Figure 11, illustrates a pop up window that data subjects see when they attempt to define such rules for a specific data controller ("The Guardian" in this example). Notably, PDGuard's data type hierarchy allows data subjects to set one rule for multiple types of data via grouping.

A data subject can view which data controllers perform actions on which data types. For instance, in Figure 12, Alice observes that "The Guardian" uses three different data types, namely: her given name, her surname, and her address. By clicking on the magnifier image Alice can view all the related uses or updates that the data controller may perform on her data and the corresponding validity period. For example, in Figure 13 we see that "The Guardian" can use Alice's surname for analytics and reporting from February 26, 2017 to March 8 2019.

Data subjects can monitor all the actions that the various applications perform on their personal data, through the authorization logs that the escrow agent provides. Figure 10 illustrates all the actions that were performed on the personal data of Alice, by The Guardian's "frontend" application, for a specific period of time. The logs also include the interaction purpose the date and the time that the action took place. Finally, the data subject can check if the action was permitted or not.

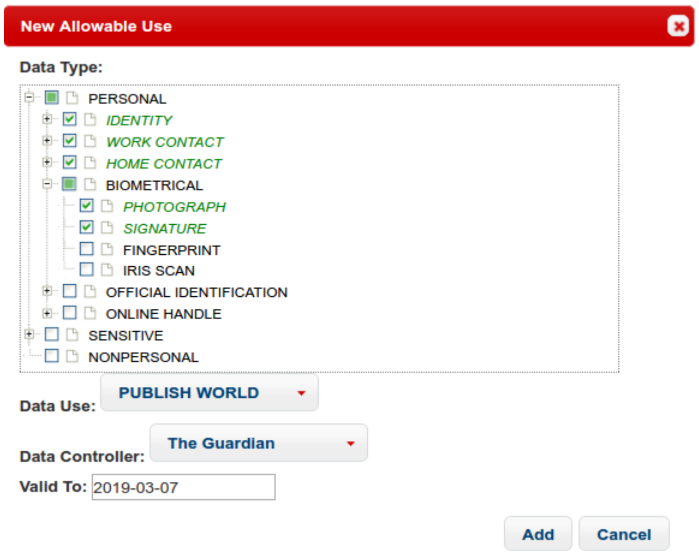


Authorization Logs

Data Controller	Application	Data Type	Data Use	Interaction Purpose	Data Provenance	Update Field	Type	Date	Result
The Guardian	frontend	Given name	-	-	Data subject explicit	true	Update	21:35 2017-02-20	Allowed
The Guardian	frontend	Surname	Report	Informative	-	-	Read	21:36 2017-02-20	Allowed
The Guardian	frontend	Home street address	Report	Informative	-	-	Read	21:36 2017-02-20	Denied by data subject
The Guardian	frontend	Given name	Report	Informative	-	-	Read	21:36 2017-02-20	Allowed
The Guardian	frontend	Home street address	Report	Informative	-	-	Read	21:36 2017-02-20	Denied by data subject

Select period:
From 2017-02-01 to 2017-02-28

Fig. 10: **Authorization Logs.** Alice monitors the different actions that "The Guardian" performed on her personal data between 2017-02-01 and 2017-02-28. Specifically, the "frontend" application sent five requests to the escrow agent. One call concerned the update of Alice's given name. This update came from Alice herself. The other requests involved calls for different data types. Note that, the escrow agent granted access only to her given name and surname.



New Allowable Use

Data Type:

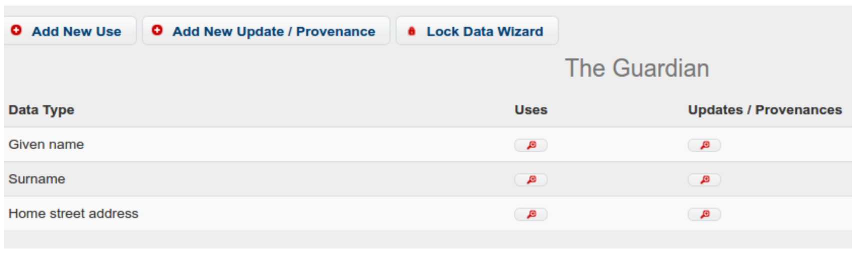
- PERSONAL
  - ✓ *IDENTITY*
  - ✓ *WORK CONTACT*
  - ✓ *HOME CONTACT*
  - BIOMETRICAL
    - ✓ *PHOTOGRAPH*
    - ✓ *SIGNATURE*
    - ☐ FINGERPRINT
    - ☐ IRIS SCAN
  - ☐ OFFICIAL IDENTIFICATION
  - ☐ ONLINE HANDLE
- ☐ SENSITIVE
- ☐ NONPERSONAL

Data Use: PUBLISH WORLD

Data Controller: The Guardian

Valid To: 2019-03-07

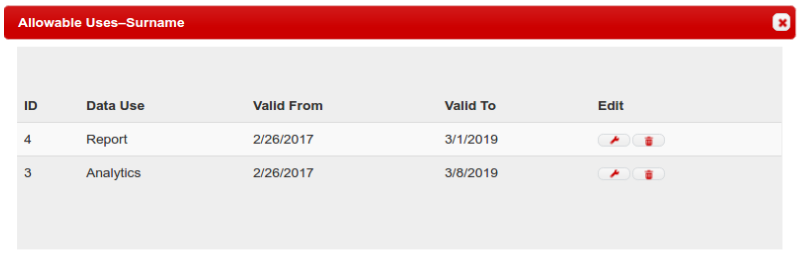
Fig. 11: **Setting Rules**. An example of the pop up window that data subjects see when they attempt to define authorization rules for a specific data controller. Here, Alice specifies which of her data will be physically published in widely available material by "The Guardian". She also specifies an expiration date for this rule.



The Guardian

Data Type	Uses	Updates / Provenances
Given name	🔍	🔍
Surname	🔍	🔍
Home street address	🔍	🔍

Fig. 12: **Data Types and Related Data Controllers.** Data subjects can observe which data types are stored by the various data controllers. In this case Alice can see that "The Guardian" stores her given name, her surname and her address. By pressing the magnifier image, Alice can see all the related uses or updates that the data controller may perform on her data and the corresponding validity periods (see also Figure 13).


Allowable Uses–Surname

ID	Data Use	Valid From	Valid To	Edit
4	Report	2/26/2017	3/1/2019	
3	Analytics	2/26/2017	3/8/2019	

Fig. 13: **Overview of Allowable Uses**. Alice views the list of the allowable uses that can be performed on her surname. Currently, the corresponding data controller can use Alice's surname for analytics and reporting. Both rules are valid until 2019. Note that, Alice can revoke or edit them.