Vinh Thong Ta* and Max Hashem Eiza

DataProVe: Fully Automated Conformance Verification Between Data Protection Policies and System Architectures

Abstract: Privacy and data protection by design are relevant parts of the General Data Protection Regulation (GDPR), in which businesses and organisations are encouraged to implement measures at an early stage of the system design phase to fulfil data protection requirements. This paper addresses the policy and system architecture design and propose two variants of privacy policy language and architecture description language, respectively, for specifying and verifying data protection and privacy requirements. In addition, we develop a fully automated algorithm based on logic, for verifying three types of conformance relations (privacy, data protection, and functional conformance) between a policy and an architecture specified in our languages' variants. Compared to related works, this approach supports a more systematic and fine-grained analysis of the privacy, data protection, and functional properties of a system. Our theoretical methods are then implemented as a software tool called DataProVe and its feasibility is demonstrated based on the centralised and decentralised approaches of COVID-19 contact tracing applications.

Keywords: privacy, GDPR, formal verification, security

DOI 10.2478/popets-2022-0028
Received 2021-05-31; revised 2021-09-15; accepted 2021-09-16.

1 Introduction

The General Data Protection Regulation (GDPR) [1] specifies the rights of living individuals who have their personal data processed, and enforces responsibilities for the data controllers and the data processors who store, process or transmit such data. Despite the data protection laws, there were several data breaches incidents in the past (e.g. [2–4]) and recently, such as the Cambridge Analytica scandal of Facebook [5], where personal data of more than 87 millions Facebook users has been collected and used for election campaign purposes without a clear data usage consent, due to the insufficient check of the third party applications.

The GDPR took effect in May 2018, and hence, designing compliant data protection policies and system architectures became even more important for organisations to avoid penalties. Data protection by design, under Article 25 of the GDPR [6], requires the design of data protection measures into the development of business processes of service providers. User profiling is restricted and businesses are required to collect consents before personal data collection (Article 6 [7]).

Unfortunately, in textual format, data protection principles are sometimes ambiguous and manual conformance verification can be error-prone. From the technical perspective, to the best of our knowledge, only a limited number of studies in the literature has addressed formal methods to design and verify policies and architectures in the context of data protection and privacy (e.g. [8–16]). The advantage of using formal methods during system design is that data protection properties can be mathematically proved, and design flaws can be detected at an early stage. Fully automated verification in this context is also challenging since we need to identify appropriate abstractions of the laws.

In this paper, we focus on early stages of system design, specifically, the policy and system architecture design stages. We show how to model and automatically verify some core and basic data protection requirements of the GDPR against a system architecture with regards to the data collection, usage, storage, deletion, and transfer phases. Core privacy requirements are also considered such as the right to access certain data, and the right to link certain pairs of data of given types. For this purpose, we propose formal methods to specify privacy policies and system architectures. In addition, we design and implement a fully automated algorithm, for verifying the conformance between a formally specified privacy policy and architecture. Specifically, our main contributions are as follows:

***Corresponding Author: Vinh Thong Ta:** Edge Hill University, Ormskirk, UK, E-mail: tav@edgehill.ac.uk
Max Hashem Eiza: Liverpool John Moores University, Liverpool, UK, E-mail: m.hashemeiza@ljmu.ac.uk

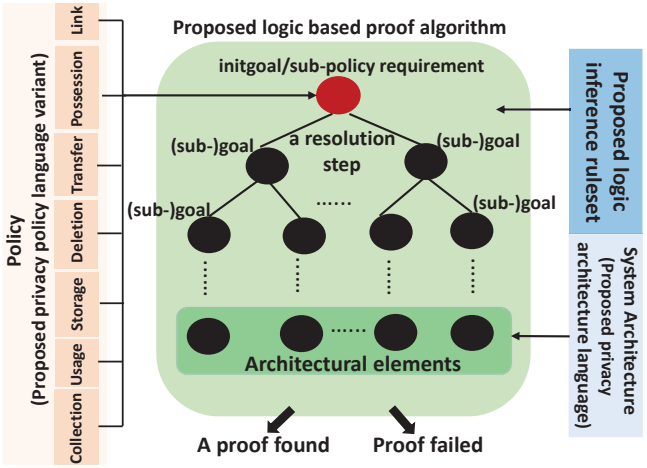

Fig. 1. A technical overview and the connection of our four main contributions. A policy that covers seven sub-policies (data collection, usage, storage, deletion, transfer, possession and link) is specified in our policy language variant. Each requirement in a sub-policy is mapped to a logic goal. The verification engine attempts to prove each goal based on a set of logic inference rules and architectural elements (defined in our architecture language). The proposed verification algorithm is based on a series of resolution steps, represented as a derivation tree, where the root is a goal to be proved, and the leaves are the architectural elements used to prove the goal.

1. We propose two variants of a privacy policy language and an architecture description language, respectively, designed for fine-grained specification of data protection and privacy requirements.
2. We propose the definition of three conformance relations between a policy and an architecture namely privacy, data protection, and functional conformance relations.
3. We propose an efficient logic based fully automated conformance verification procedure for verifying the three types of conformance relations between a policy and an architecture specified using our proposed languages' variants.
4. Finally, we develop a (prototype) tool, DataProVe, based on the proposed automated conformance verification procedure, and demonstrate its usage on real-world COVID-19 contact tracing approaches.

The technical overview of our contributions is outlined in Figure 1. The main goals of our policy and architecture languages' variants and DataProVe include: 1) helping system designers with policy and architecture design and analysis; and 2) to spot potential errors early, prior to the lower level (such as the protocol or code level) system specification and implementation.

The paper is structured as follows: In Section 2, we discuss the related works in the literature. In Sections 3.1-4, we present the scope of this work and our variants of privacy policy and architecture description languages, respectively. The automated conformance verification engine is detailed in Section 6. In Section 7, we present the DataProVe tool and its operation on two COVID-19 contact tracing approaches. In Section 8, we discuss performance issues. Finally, we discuss the future directions and conclude the paper in Sections 9-10.

2 Related Works

In this section, we highlight the most relevant works in the field and compare them with our work in Section 2.1. The Platform for Privacy Preferences (P3P) [17] enables web users to gain control over their private information on online services. On a website, users can express their privacy practices in a standard format that can be retrieved automatically and interpreted by web client applications. Users are notified about certain website's privacy policies and have a chance to make a decision on that. To match the privacy preferences between users and web services, the authors proposed the Preference Exchange Language (APPEL) [18], which is integrated into the web clients. In APPEL, users can express their privacy preferences that can be then matched against the practices set by the online services. According to the study [19], in APPEL, users can only specify what is unacceptable in a policy. To rectify this, the authors in [19] proposed a more expressive preference language called XPref that also supports acceptable preferences.

A-PPL [9] is an accountability policy language specifically designed for modelling data accountability (such as data retention, logging and notification) in the

cloud. A-PPL is an extension of the PrimeLife Privacy Policy Language (PPL) [8], which enables specification of access and usage control rules for the data subjects and the data controller. PPL is built upon XACML, and allows users to define the so-called sticky policies on personal data based on obligations. An obligation defines whether the policy language can trigger tasks that must be performed by a server and a client, once a specific event occurs and the related condition is fulfilled.

In [20], the authors extended the Unified Modeling Language (UML) meta model to specify and represent different activities on data that can be checked for privacy compliance. The operations of the data life-cycle were modelled using the data flow diagrams (DFDs) with relevant actors, operations, relationships and conditions. It is unclear how this model can be used in the context of designing system architectures that are in compliance with privacy policy given the complexity of modelling UML DFDs. Besides that, the authors did not offer an automated approach to verify compliance with the data privacy policy.

Celebi [10] proposed a new modelling language called Privacy Enhanced Secure Tropos (PESTOS) along with its meta-model, semantics and concrete syntax. PESTOS can aid developers catching GDPR privacy requirements at an early stage during their system design. Its limitations include lack of automated check for privacy conformance, and the architecture level.

The authors in [21] presented a set of privacy-related architectural requirements, and showed how they may be implemented in practice in the context of FIM (federated identity management) models. Eight requirements (PP1-8) were taken from the EU Directive 95/46/EC, and mapped into five design requirements (FDR1-5) in the context of FIMs. These include FDR1 (limit the disclosure of identity), FDR3 (make illegitimate linking of data difficult), FDR4 (transparency) and FDR5 (information security). The authors also presented architectural requirements for FIM (AR1-AR8), for example, AR1-2 specify limited observability and linkability with regards to data aggregation across services. AR3-4 deal with limiting the data collection centrally, and linkability, while the rest address the consent handling, data minimisation, and unique linkability.

In [13], the authors proposed a formal approach to model the transmission of personal data. Their policy specification focuses on the transmission of information, and also follows a role-based model. First-order temporal logic is used to express contextual integrity requirements at the policy level. A communication is allowed between agents if either its temporal condition is satisfied (positive norm), or unsatisfied (negative norm). The authors defined the following logic formulas for policy requirements: **send**(p1, p2, m), which captures when agent p1 sends p2 a message m; **contains**(m, q, t), and **inrole**(p, r) that capture a message m contains an element, and agent p is in a role r, respectively. They also define **incontext**(p, c) for contextual integrity.

The work in [14] addresses the policy enforcement on codes during system operations in the context of big data within an organisation such as Microsoft. In their policy specification, users can specify requirements on which data types are allowed/denied to be used for which purposes. However, the strength of their approach relies on the verification of codes against a policy. In particular, they address codes written in languages that support the Map-Reduce programming model (big data context). They address the code level, hence, concentrate on one specific application, such as the data analytics backend of Microsoft Bing. Their automated verification is based on automated data-inventory mapping, and has two elements: (1) labelling the data types in a code, based on the labelled data dependency graphs, (2) then check the labelled data types against the policy.

In [15], a systematic methodology to model privacy-specific threats for threat analysis, focusing on software-based systems, is proposed. Their approach relies on data flow diagrams (DFDs), which can be seen as high-level reasoning. In a DFD, data flows model communication data, and trusted/untrusted agents are also defined to identify threats. This approach is good for addressing large and complex systems, and high-level threats.

Besides the academic's effort, some industrial approaches are also available such as the Open Policy Agent (OPA) project [16]. OPA focuses on the cloud-nature environment and aligns its syntax to the cloud environment. Its policy specification is based on the Rego (declarative) language, which provides an expressive way to define fine-grained policies. In OPA, the data or system specification is based on informal method in JSON (JavaScript Object Notation) format. A user-defined policy rule (in Rego) will be evaluated on the provided data/system in JSON, which generates the so-called *virtual document* that is defined by the rule. This is then used to enforce a policy during a system run.

Finally, there are numerous formal languages proposed for architectures including Darwin [22], Wright [23], Rapid[24], and PRISMA [25]. These approaches are based on process calculi (e.g. pi-calculus [26], and FSP algebra [27]), to specify dynamic architectures. They define the so-called *bindings* to model the connections between components based on process calculi semantics.

2.1 Comparison With Our Work

Table 1 shows a high-level comparison of our work and the related works based on seven aspects. The main differences between the policy languages above and our method is that, for instance, P3P [17], APPEL [18], and XPref [19] are mainly designed for web services, and the policies are defined in an XML-based language, with restricted options for the users, while our design is not limited to web services. In addition, our policy language variant is defined on data types, and supports a more systematic policy specification, as its syntax and semantics cover seven sub-policies capturing a representative data life-cycle (from collection till deletion).

Some requirements in paper [21] are addressed in our work. For example, PP2 and PP3 are addressed in our sub-policies with regards to purposes, and PP5, AR2-3 are addressed by our data possession policy and cryptographic model. The requirements PDR1, FDR3, AR4 and AR8 are addressed with our linkability/unique linkability sub-policies. Finally, AR5 is addressed with our consent collection requirements.

In OPA [16], the user has the freedom to specify the policy rules, but they need expertise in Rego. On the other hand, in our tool, the user can specify a policy without special knowledge on a declarative language. In addition, their policy specification supports different requirements compared to our seven sub-policies (e.g. “servers are not allowed to expose the 'telnet' protocol"). It is unclear how linkability and the attackers' related requirements (such as an insider/external attacker can have/link certain data types) could be defined with their policy language. In addition, a system specified in JSON is quite abstract, as unlike our method, it only specifies the components and the properties of the components, but not the communications (e.g. RECEIVE, in our case) and none of our CALCULATE, DELETE, STORE, etc., actions. The philosophy of OPA is merely different from ours, as we address the (early) design phase, while they address the policy enforcement during a system run. Finally, the verification is based on the query language Datalog. The advantage of our “crafted" verification engine compared to Datalog is the formal cryptographic model, and the attacker model in which the attacker can eavesdrop on the communication.

Compared to [13], the **send**(p1, p2, m) formula is addressed by our RECEIVE action to capture data transmission, while **contain** is modelled, in our case, by the definition of “compound" data types that contain other data types as arguments. Their **inrole**(p, r) and **incontext**(p, c) formulas for role-based access control and contextual integrity are not considered in our approach. On the other hand, their paper did not propose any formal architecture language, and their approach lacks formalism for the data possession, purposes, linkability requirements, as well as automated verification. Unlike our work, paper [15] does not address the policy level but only architectures nor addresses automation verification. The philosophy of their threat based approach is different from ours, but it can be used alongside our method to improve the system design.

	Pol. Level	Arch. Level	Auto Verif	Crypto	Code Level	Threat Model	Attacker
Ours	**Y(F)**	**Y(F)**	**Y(F)**	**Y(F)**	-	-	**Y(F)**
[...]	**Y(F)**	-	-	-	-	-	-
[21]	**Y(I)**	**Y(I)**	-	-	-	-	-
[13]	**Y(F)**	-	-	-	-	-	-
[15]	-	**Y(I)**	-	-	-	**Y(I)**	-
[14]	**Y(F)**	-	**Y(F)**	-	**Y(F)**	-	-
[16]	**Y(F)**	-	**Y(F)**	-	**Y(I)**	-	-

Table 1. In the table, Y(F) means *“The framework supports formal specification of a given level or feature"*, while Y(I) means *“It supports only informal specification"*. The works in [...] are [8–10, 17–20], and Pol. = Policy, Arch. = Architecture.

Similarly, the philosophy of the work in [14] is different from ours, as we address the architecture level at the design phase. With the architecture language variant we propose, one can reason about a service that involves different organisations with their components communicating with each other. At the code level, it would be very complex to extend the analysis to the entire service with many components interacting with each other across different organisations. In addition, their policy specification does not support data linkability and data possession. Finally, there is neither support for a cryptographic model nor the reasoning about the attackers.

Furthermore, unlike the Architecture Description Languages (ADLs) in [22–25, 28], our ADL variant is designed to capture the data protection and privacy properties. It also supports cryptographic primitives and enables attacker modelling. Our architecture language variant is *data type* centred, and its semantics does not rely on process algebra like the ADLs in [22–25, 28], but is based on events and event traces. This concept was applied in some previous papers, such as in [12, 29]. The language variants in [12, 29] mainly focus on the computation and integrity verification of data based on trust relations. Our work is inspired by [12, 29], but unlike them, our proposed architecture language variant focuses on data protection and privacy properties.

3 The Privacy Policy Level

In this section, we introduce the proposed policy language variant for specifying privacy policies. We discuss the scope and limitations of our design in Section 3.1 and present the policy language in Section 3.2.

3.1 Scope and Limitations

Due to space constraint, we mainly focus on the GDPR rules capturing the core mandates that are also shared by several other regulations. The relevancy of our chosen rules is due to the fact that the data access and linkability requirements are addressed in many privacy laws, while the satisfaction of the consent collection, collection and storage purposes, retention delay, data storage and transfer requirements are shared by other laws such as the Nigeria Data Protection Regulation (NDPR) [30], the California Consumer Privacy Act (CCPA) [31], and the Personal Information Protection and Electronic Documents Act (PIPEDA, Canada) [32].

As will be seen in the paper, our method is expressive and can be used to verify complex real-life services efficiently; however, it does not cover the whole GDPR. Our policy language covers seven sub-policies capturing the core mandates of the GDPR, but for example, it does not support the conditional and declarative constructs, which are required by some complex rules (e.g. *Article 46(1)* [33], which states that data transfer to a country outside the EU is only allowed if certain conditions are met). Our transfer sub-policy only includes a set of third-parties to whom personal data can be transferred (we assume that the conditions are already met).

Due to the challenge with formalising laws, we focus on the GDPR requirements that require moderate abstraction for formalisation and automation. For instance, the consent requirements we model in the current framework is based on the fact of whether a consent collection happens or not. In case of purposes, we address the verification of whether an architecture conforms with the set of purposes defined in the policy or not (which is covered by *Article 6(1)(a)*) [7]. However, we neither consider the children consent issue which involves the role of a guardian (*Article 6(1)(f)* [7]), nor the requirement on the "vital interest of the data subject" (*Article 6(1)(b)* [7]), which would need further investigation due to their complexity. Finally, we do not consider non-numerical retention delays for which the verification requires further abstractions.

3.2 The Policy Language

Our proposed privacy policy is defined from the perspective of a data controller, who we assume to be a service provider that collects, stores, uses or transfers the personal data of the data subjects. It covers both (i) data protection and (ii) privacy requirements. The data subjects in our case are service users whose personal data is collected and used by the data controller.

A privacy policy is defined on a data type θ (e.g. θ = bill). Specifically, let π_θ be a policy defined on a data type θ, and consists of seven sub-policies. Namely,

$$\pi_\theta = (\pi_{col}, \pi_{use}, \pi_{str}, \pi_{del}, \pi_{fw}, \pi_{has}, \pi_{link}).$$

Each sub-policy of π_θ is defined as follows:

1. Data collection sub-policy $\pi_{col} = (cons, cpurp)$, where $cons \in \{Y, N\}$ specifies whether a consent is required to be collected from the data subject (Y) or not (N) for a data type θ, and *cpurp* is a set of collection purposes. A purpose has the form $act_i{:}\theta_i$, which specifies that a piece of data of type θ is collected by the service provider to perform an action act_i to get some data of type θ_i. For instance, θ = *name* is collected to *create* an *account* (i.e. the purpose is *create*:*account*). These aim to capture the consent and purposes limitation requirements in *Article 6* [7] and *Article 5(1)(b)* [34] of the GDPR.
2. Data usage sub-policy $\pi_{use} = (cons, upurp)$, with a usage consent requirement, $cons \in \{Y, N\}$, and *upurp*, a set of usage purposes. These capture the *Article 6* [7] and *Article 30(1)(b))* [35] of the GDPR.
3. Data storage sub-policy $\pi_{str} = (cons, where)$, in which *where* is a set of places where a piece of data of type θ can be stored (e.g. in a client's machine *where* = {*decentralised*} while on the service provider's main and/or backup storage *where* = {*mainstorage*}, *where* = {*backupstorage*}). These elements partly capture the storage limitation principle in *Article 5(1)(e)* [34] of the GDPR.
4. Data deletion sub-policy $\pi_{del} = (fromwhere, deld)$:
   - *fromwhere* contains the locations from where a piece of data can be deleted. This strongly depends on the storage locations, *where*, which is defined in the storage sub-policy π_{str}.
   - *deld* is the delay value for retention. This value can be a specific numerical time value (e.g. 1 day (1d), 10 mins (10m), 5 years (5y)).

   These two elements partly capture the *Article 5(1)(e)* and *Article 17(1)(a,e)* [36] of the GDPR.

5. Data forward sub-policy $\pi_{fw} = (cons, fwto, fwpurp)$, where *cons* specifies the requirement for the data transfer consent, *fwto* specifies a set of entities to whom the data can be transferred to, and *fwpurp* is a set of purposes for the data transfer. These partly capture the requirement of transferring data to third-party organisations in *Article 45(1)* [37].
6. Data possession sub-policy $\pi_{has} = whocanhave$ addresses information security, where $whocanhave = \{E_1,\ldots, E_k\}$ is a set of entities involved in delivering the service, which have the right to have or possess a piece of data of type θ. By default, no one has the right to have any type of data.
7. Data connection sub-policy $\pi_{link} = whocanlink$, where $whocanlink = \{(E_1,\theta_1),\ldots, (E_m,\theta_m)\}$ is a set of pairs of entities and data types. Each pair (E_i, θ_i) specifies that E_i has the right to link a data type θ_i to the data type θ. For instance, if a service provider *sp* has the right to link a piece of information about someone's disease with their work place, then $E_i = sp$, $\theta_i = disease$, $\theta = workplace$.

Finally, let $\{\theta_1, \ldots, \theta_m\}$ be a set of all data types used by the service of a service provider *sp*. The privacy policy of *sp* is defined by: $\mathcal{PL} = \{\pi_{\theta_1}, \ldots, \pi_{\theta_m}\}$. The semantics of the policy language is highlighted in Appendix F.

4 The Architecture Level

System architectures describe how a system is composed of entities and how they relate to each other. These details are abstracted away from the policy level.

4.1 ADL Variant: The Proposed Syntax

In line with the policy specification, a system architecture is defined on a set of entities and data types. For a service *service*, let $EntitySet_{arch}^{service} = \{E_1, \ldots, E_k\}$ be a finite set of entities, and $DataTypes_{arch}^{service} = \{\theta_1,\ldots, \theta_m\}$ the set of all data types defined in an architecture. Moreover, we assume the finite sets of data variables *Var*, ($X_\theta \in Var$), time variables *TVar* ($TV \in TVar$). Finally, we define the finite sets of the data values *Val* ($V_\theta \in Val$), the time values *TVal* ($t \in TVal$), and deletion delay values *DVal* ($dd \in DVal$).

Terms: A term, denoted by T, is defined as:

$$T ::= X_\theta \mid V_\theta \mid ds \mid E \mid SpecFunc \mid Ti.$$

Specifically, a term can be one of followings:

- X_θ, which is a variable that can be some non-function data D_θ of type θ, a cryptographic function (*CryptoFunc*), or any service specific function.

$$X_\theta ::= D_\theta \mid CryptoFunc \mid Serv\_spec\_fun(X_{\theta_1}, \ldots, X_{\theta_n}).$$

- V_θ, which is a data value of type θ (e.g. θ=bill, V_{bill} = 9USD).
- *ds*, which is a special term that specifies the real identity of a data subject (this will be used for modelling pseudonyms).
- E, which is an entity term that specifies any software or hardware component, organisations, devices, etc. (e.g. $E = phone$, or *server*, or *thirdparty*).
- *SpecFunc*, which can be the time function, the pseudonym function, or four functions that specify the four types of consents as explained later.

$$\begin{aligned} SpecFunc ::= {} & \mathbf{Time}(Ti) \mid \mathbf{P}(ds) \mid \mathbf{Cconsent}(Data) \mid \\ & \mathbf{Uconsent}(Data) \mid \mathbf{Sconsent}(Data) \\ & \mid \mathbf{Fwconsent}(X_\theta, E_{to}) \mid \mathbf{Meta}(X_\theta). \end{aligned}$$

- Ti, where $Ti ::= dd \mid TT$, is a time value that can be either a numerical delay value (*dd*, e.g. *dd*=5s), or a non-specific value (TT).

Special functions: As mentioned before, there are two groups of functions *SpecFunc* and *CryptoFunc*. We start by explaining *SpecFunc* as the following:

- Function **Time**(Ti), as explained earlier, specifies the time with either a non-specific time value TT or a numerical delay value, *dd*. While *dd* captures numerical time values such as 3 years (3y), 5 days (5d), etc., TT is not numerical, and is used to express the informal term "at some point/time".
- **P**(*ds*) specifies a pseudonym of a real identity *ds*.
- Functions **Cconsent**(*Data*), **Uconsent**(*Data*) and **Sconsent**(*Data*) specify the consent needed for collection, usage and storage, respectively, on *Data* that can be expressed as follows:

$$Data ::= (X_\theta, E_{consent})$$

  where $E_{consent}$ is an entity who can do the required action on the data X_θ (as part of the consent). Finally, **Fwconsent**(X_θ, E_{to}) specifies a transfer consent on data X_θ, alongside an entity to whom the data can be transferred to (E_{to}).
- Function **Meta**(X_θ) defines some metadata (information about other data), or the information located in the header of the packets (e.g. IP address).

Cryptographic model: We formalise cryptographic primitives and operations based on cryptographic func-

tions and the so-called destructor. This approach provides an abstract representation of cryptographic primitives and operations and has been applied in cryptographic protocol verifiers (e.g. ProVerif). *CryptoFunc* captures the basic cryptographic functions as follows:

- **Sk**($X_{pkeytype}$) function defines the type of private key used in asymmetric key encryption algorithms. It has a public key $X_{pkeytype}$ as argument.
- **Senc**(X_θ, $X_{keytype}$) defines the type of a symmetric key encryption (cipher text), and has two arguments, a piece of data (of type θ) and a symmetric key (of type *keytype*).
- **Aenc**(X_θ, $X_{pkeytype}$) defines the type of the cipher text resulted from an asymmetric key encryption, and has two arguments, a piece of data and a type of public key $X_{pkeytype}$.
- **Mac**(X_θ, $X_{keytype}$) defines the type of the message authentication code that has two arguments, a piece of data and a symmetric key.
- **Hash**(X_θ) defines the type of the cryptographic hash that has a piece of data as argument.

Decryption/verification: The decryption and verification of cryptographic functions can be defined by the following function: $G(T_1, \ldots, T_n) \rightarrow T$.

For instance, if $X_{enc} = Senc(X_{name}, X_{Skey})$ represents the encryption of X_{name} with the server key X_{Skey}, and X_{Skey} represents a symmetric key, then $G(X_{enc}, X_{Skey}) \rightarrow X_{name}$ is $Dec(Senc(X_{name}, X_{Skey}), X_{Skey}) \rightarrow X_{name}$. Note that not all functions have a corresponding destructor. For instance, X_{hash} is a one-way cryptographic hash function $X_{hash} = Hash(X_{password})$, due to the one-way property, there is no destructor (reverse procedure) that returns $X_{password}$ from the hash.

This assumes "perfect" cryptography primitives, namely, only the entity who has the key can access the plaintext. Currently, we have not considered the case of partial information leakage or guessing, which can be important to capture privacy violation. Some formal models for information leakage can be found in process calculi (e.g. the applied pi-calculus) based on observational equivalence. Our approach is extendable, and also supports the abstract modelling of complex concept such as homomorphic encryption (see Appendix B).

4.2 The Definition of an Architecture

An architecture $\mathcal{PA}$ is defined as a set of *actions* (denoted by $\{\mathcal{F}\}$), with the following formal definition:

$\mathcal{PA} ::= \{\mathcal{F}\}$
$\mathcal{F} ::=$
$\quad OWN(E, X_\theta)$
$\quad \mid CALCULATEAT(E, X_\theta, \textbf{Time}(TT))$
$\quad \mid CREATEAT(E, X_\theta, \textbf{Time}(TT))$
$\quad \mid RECEIVEAT(E, X_\theta, \textbf{Time}(TT))$
$\quad \mid RECEIVEAT(sp, \textbf{ConsentType}, \textbf{Time}(TT))$
$\quad \mid STOREAT(E, X_\theta, \textbf{Time}(TT))$
$\quad \mid DELETEWITHIN(E, X_\theta, \textbf{Time}(dd))$.

Fig. 2. The definition and the syntax of a system architecture. *sp* is a value that denotes the service provider. **ConsentType** can be collection - **Cconsent**(*Data*), usage - **Uconsent**(*Data*), storage - **Sconsent**(*Data*), and forward - **Fwconsent**(X_θ, E_{to}). *Data* = (X_θ, $E_{consent}$), where $E_{consent}$ is an entity who is given consent to receive/calculate/create/store X_θ, and E_{to} is to whom data X_θ can be transferred.

1. Action $OWN(E, X_\theta)$ captures that E can own the data X of type θ (and is aware of its properties).
2. $CALCULATEAT(E, X_\theta, \textbf{Time}(TT))$ specifies that an entity E can calculate the variable X_θ based on an equation $X_\theta = T$, for a term T at a non-specific time TT (e.g. θ = *Bill*(*energy, tariff*), and X_θ = {34kWh, 65USD}).
3. $CREATEAT(E, X_\theta, \textbf{Time}(TT))$ specifies that E can create a piece of data of type θ, based on an equation $X_\theta = T$ (e.g. θ = *Account(name, ID), and* X_θ *can be {Peter, 1234}*). The actions *create* and *calculate* merely differ in the nature of T, for example, we calculate a bill, while create an account.
4. $RECEIVEAT(E, X_\theta, \textbf{Time}(TT))$ specifies that E can receive X_θ at some non-specific time TT.
5. $RECEIVEAT(sp, \textbf{ConsentType}, \textbf{Time}(TT))$ specifies that a consent for collecting, usage, storage, or transfer of data X_θ can be received by the service provider *sp* at a non-specific time TT.
6. $DELETEWITHIN(E, X_\theta, \textbf{Time}(dd))$ specifies that X_θ must be deleted from a place E within a certain time delay dd (where dd is a numerical time value, e.g. dd = 10y (10 years)).

These actions, to the best of our knowledge, can be found in many services, and pose high privacy risk. The choice of the names *RECEIVEAT*, *CALCULATEAT*, etc., besides being intuitive, is very useful for automated reasoning, where the actions need to be translated into the corresponding logic facts. Our proposed presentation of the actions can be directly translated to the logic facts in a very straightforward way. For instance, an action *RECEIVEAT(EntityTerm, DataTerm, TimeTerm)* can be translated to the logic

fact *RECEIVEAT(EntityVar, DataVar, TimeVar)*. In addition, the actions *store* and *delete* are known to be complicated to formalise. In our case, these actions can be translated to the logic facts *STOREAT* and *DELETEWITHIN* straightforward. Note that there are services which require more actions (besides ours) to model (e.g. actions *STRUCTURE*, *ORGANISE*, *RECORD*). Our architecture language can be extendable with these additional service specific actions. Finally, the semantics of the architecture language is highlighted in Appendix G.

An example architecture is shown in Figure 3 about a smart meter service. In this service, the server can receive an energy record with the gas and electricity readings at some non-specific time, and this record can be stored in the main storage place(s) of *sp*.

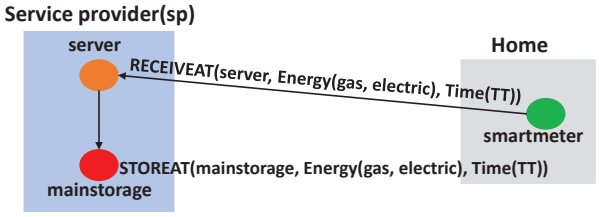

Fig. 3. A simple smart meter architecture. For data type θ, we have θ = Energy(gas, electricity), while the data X_θ can take the value {10kWh, 12kWh} during an instance of the system run.

5 The Conformance Between Policies and Architectures

We propose three types of conformance relations: (i) privacy conformance, (ii) conformance with regards to data protection properties (which we refer to as DPR conformance in this paper), and (iii) functional conformance. Below, we provide the definition of each.

Privacy conformance compares a policy and an architecture based on the privacy properties. Specifically, if we do not give an entity the right to have certain type of data or link two types of data, then in the architecture this entity cannot have or link those types of data.

Definition 1. *(Proposed privacy conformance)*

1. *If in a policy π_θ an entity E does not have the right to have any data of type θ, then E cannot have this type of data in the corresponding architecture.*
2. *If in a policy π_θ an entity E does not have the right to link two types of data, θ_1 and θ_2, then E cannot link these in the corresponding architecture.*

The DPR conformance relation deals with the data protection requirements (specified in the first five sub-policies), such as appropriate consent collection, the satisfaction of the defined deletion/retention delay and the appropriate storage and transfer of a given type of data.

Definition 2. *(Proposed DPR conformance)*

1. *If in a policy π_θ, the collection of a (collection, usage, storage, or transfer) consent is required for a piece of data of a given type, then in the architecture the reception of a consent happens before or at the same time with the reception of the data itself.*
2. *If in an architecture, there is an action **act** (createat or calculateat) defined on a type θ of data, then in the policy π_θ, there is a (collection, usage, storage, or transfer) purpose **act**:θ' defined for the data type θ (for some type θ' as the result of the action **act**).*
3. *If in an architecture, a piece of data of type θ can be stored in some storage place, **strplace**, then in the policy π_θ, **strplace** $\in \pi_{str}$.**where***
4. *If in the policy π_θ, **delplace** $\in \pi_{del}$.**fromwhere**, then in the corresponding architecture the data of type θ can be deleted from the place **delplace**.*
5. *If in an architecture, a piece of data of type θ can be deleted within a delay **dd** (since collection), then in the corresponding policy π_θ, **dd** $\leq \pi_{del}$.**deld**. In other words, the retention delay defined in the policy must be respected in the architecture.*
6. *If in an architecture, a piece of data of type θ can be transferred to an entity E_{to}, then in the policy π_θ, $E_{to} \in \pi_{fw}$.**fwto**.*

Finally, functional conformance compares a policy and an architecture from the perspective of functionality or effectiveness. This conformance can help a system designer to find an appropriate trade-off between functionality and privacy to provide certain services.

Definition 3. *(Proposed functional conformance)*

1. *If in a policy π_θ, an entity E has the right to have a type of data, θ, then E can have this type of data in the corresponding architecture.*
2. *If in a policy π_θ, an entity E has the right to link two types of data, θ_1 and θ_2, then E can link these types of data in the corresponding architecture.*
3. *If in a policy π_θ, the collection of a (collection, usage, storage, or transfer) consent is **not** required,*

then ***no*** *corresponding consent can be received in the corresponding architecture.*

4. *If in a policy π_θ, there is a (collection, usage, storage, or transfer) purpose* ***act***$:\theta'$ *defined, then in the corresponding architecture there is an action* ***act*** *defined on a data type θ (for some θ').*
5. *If in a policy π_θ, (****strplace*** $\in \pi_{str}.$***where****) for some storage place* ***strplace****, then in the architecture the data of type θ can be stored in* ***strplace****.*
6. *If in an architecture, a piece of data of type θ can be deleted from a storage place,* ***delplace****, then in the corresponding policy π_θ, we have (delplace $= \pi_{del}.$****fromwhere****).*
7. *If in the policy π_θ, $E_{to} \in \pi_{fw}.$****fwto****, then in the architecture, the data of type θ can be transferred to the same entity E_{to}.*

6 The Automated Conformance Verification Procedure

We apply the (logic) resolution based proof concept (widely used in semi-automated theorem provers) in a new context, and crafted the proof algorithm to make it fully automated, and designed the inference rules specifically for the privacy, DPR and functional conformance.

6.1 The Inference Rules Used in the Proof

Definition 4. *An inference rule R is denoted by $R = H \vdash F_1, \ldots, F_n$, where H is the head of the rule and $F_1, \ldots, F_n$ is the tail of the rule. Each element F_i of the tail is called a fact, and a head is called a "consequence". The rule R reads as "if $F_1, \ldots, F_n$, then H".*

In the following, let EV, θV, TV and DD denote an entity variable, data type variable, non-specific time variable and numerical delay variable, respectively. The entity value *sp* denotes the service provider. We start with the proposed rules used in the verification of the privacy conformance relation (wrt. data possession):

Rule *P1* says that if an entity EV can store data of type θV, and can delete it within a delay, DD, then the entity can have this data up to DD time.

P1. HASUPTO(*EV*, θV, Time(*DD*)) ⊢ STOREAT(*EV*, θV, Time(*TV*)), DELETEWITHIN(*EV*, θV, Time(*DD*)).

Rule *P2* says that if a trusted authority or organisation has any data that contains a pseudonym ($\mathbf{P}(ds)$), alongside some other data, then the trusted authority (*trusted*) can also have the same data that contains the "real" identity *ds*.

P2. HAS(trusted, Anytype(*ds*, θV)) ⊢ HAS(trusted, Anytype(θV, P(*ds*))).

Where Anytype $\notin \{Senc, Aenc, Mac, Hash\}$). In addition, *trusted* and *ds* are constant values.

Rule *P3* says that if EV can own a type of data (regardless of time), then it can have this data.

P3. HAS(*EV*, θV) ⊢ OWN(*EV*, θV).

Rule *P4* says that if EV can receive θV at some non-specific time TV, then it can have this data. Rules *P5*-*P7* can be interpreted in a similar way.

P4. HAS(*EV*, θV) ⊢ RECEIVEAT(*EV*, θV, Time(*TV*)).
P5. HAS(*EV*, θV) ⊢ STOREAT(*EV*, θV, Time(*TV*)).
P6. HAS(*EV*, θV) ⊢ CALCULATEAT(*EV*, θV, Time(*TV*)).
P7. HAS(*EV*, θV) ⊢ CREATEAT(*EV*, θV, Time(*TV*)).

Rules *P8*-*P9* capture the decryption/verification of the cryptographic data types. *P8* says that if EV can have the encryption of a piece of data of type θV using a symmetric key K, and it can also have K, then it can have the data. Rule *P9* deals with the asymmetric decryption process, where *Sk(PK)* denotes a private key.

P8. HAS(*EV*, θV) ⊢ HAS(*EV*, Senc(θV, *K*)), HAS(*EV*, *K*).
P9. HAS(*EV*, θV) ⊢ HAS(*EV*, Aenc(θV, *PK*)), HAS(*EV*,Sk(*PK*)).

One reason we add cryptography at the architecture level is to, for example, distinguish architectures that apply client-side encryption from the server-side encryption. **Note**: We do not have decryption rules for MAC/hash functions due to their one-way property.

Figure 4 presents the proposed rules used in the verification of the DPR conformance relations: *D1* specifies that if the service provider *sp* can receive a transfer consent on θV to EV_{to} at some non-specific time TV, and the entity (defined by the variable) EV_{to} can receive this at the same time or later (this is abstractly modelled by the same non-specific time variable TV), then we say that *sp* can collect the transfer consent on θV to EV_{to}. The rest of the rules can be interpreted in a similar way, as rule *D2* is defined for the data collection

D1. FWCONSENTCOLLECTED(sp, θV, EV_{to}) ⊢
RECEIVEAT(sp, **Fwconsent**(θV, EV_{to}), **Time**(TV)),
RECEIVEAT(EV_{to}, [θV], **Time**(TV)).
D2. CCONSENTCOLLECTED(sp, θV) ⊢
RECEIVEAT(sp, **Cconsent**($Data$), **Time**(TV)),
RECEIVEAT($EV_{consent}$, [θV], **Time**(TV)).
D3. UCONSENTCOLLECTED(sp, θV) ⊢
RECEIVEAT(sp, **Uconsent**($Data$), **Time**(TV)),
CALCULATEAT($EV_{consent}$, [θV], **Time**(TV)).
D4. STRCONSENTCOLLECTED(sp, θV) ⊢
RECEIVEAT(sp, **Sconsent**($Data$), **Time**(TV)),
STOREAT($EV_{consent}$, [θV], **Time**(TV)).

Fig. 4. The proposed inference rules for DPR conformance check. *EV*, θV, *TV* denote an entity variable, data type variable, and a non-specific time variable, respectively. **[**θV**]** denotes a type of data (except for *Senc*/*Aenc*/*Mac*/*Hash*) that contains a piece of data of type θV, or equal to θV itself.

consent, rule *D3* deals with the usage consent collection, while *D4* is related to the storage consent.

Figure 5 highlights some examples of the proposed rules for the verification of the privacy conformance relation (wrt. linkability). Specifically:

- Rule *L0* says that if an entity EV can have two pieces of data of types θV_1 and θV_2, inside any compound data types with the same metadata, then EV can link θV_1 and θV_2.
- Rule *L1* says that if an entity EV can have any data that contains two pieces of data of types θV_1, and θV_3 besides any other data (denoted by θV and $\theta V'$), and any data that contains two pieces of data of types θV_2 and θV_3, then EV can link θV_1 and θV_2. Note that this is not a "unique" linkability, meaning that EV cannot be sure that the data of types θV_1 and θV_2 belong to the same individual (although it can narrow down the set of possible individuals to some extent).
- Rule *U0* says that if EV owns θV_1 and θV_2, then it can uniquely link those (as it is aware of their nature and properties).
- Extending *L1*, rule *U1* says that if a type θV_3 is unique (e.g. passport numbers with country codes), then EV can also "uniquely" link the data of types θV_1 and θV_2, namely, it can also be sure that they belong to the same individual.

We define linkability between the arguments of compound data types. We assume that all the data types inside a compound data type belong to the same living individual. For example, in the compound data type *Bill(name, amount, address)*, name, amount, and address belong to the same individual. This assumption can be seen as a simplification; however, it holds in many situations and compound data types. We also define that if E can calculate two pieces of data D1, D2 from a piece of data D that belongs to the same individual, then E can link D1 and D2. In addition, we also define "trivial" linkability between data types that are owned by an entity as we assume that the owner is aware of the nature of the data it owns. Linkability can also be resulted from unique meta information. However, currently, we have not considered the case of linkability that is resulted from partial information leakage.

L0. LINK(EV, θV_1, θV_2) ⊢
HAS(EV, Anytype1(θV_1, θV, **Meta**(θV_3))),
HAS(EV, Anytype2(θV_2, $\theta V'$, **Meta**(θV_3))).
L1. LINK(EV, θV_1, θV_2) ⊢
HAS(EV, Anytype1(θV_1, θV, θV_3)),
HAS(EV, Anytype2(θV_2, $\theta V'$, θV_3)).
U0. LINKUNIQUE(EV, θV_1, θV_2) ⊢ OWN(EV, θV_1),
OWN(EV, θV_2).
U1. LINKUNIQUE(EV, θV_1, θV_2) ⊢
HAS(EV, Anytype1(θV_1, θV, θV_3)),
HAS(EV, Anytype2(θV_2, $\theta V'$, θV_3)), UNIQUE(θV_3).

Fig. 5. Rules for privacy conformance check (linkability, unique linkability). Where we assume that in any compound type, i.e., *Anytype*(θ_1,...,θ_k), θ_1,..., θ_k belong to the same individual.

6.2 Proposed Conformance Check Algorithm

The automated conformance verification is based on the execution of *resolution* steps and backward search. Resolution is well-known in logic programming and widely applied in logic programming languages and theorem provers. The formal definition of resolution is based on the so-called unification steps. A unifer, denoted by σ, binds/assigns a value (e.g. E, θ, TT, dd) to a variable (e.g. EV, θV, TV, DD, respectively).

Definition 5. *A unifier σ is the most general unifier of a set of facts $\mathbb{F}$ if it unifies the facts in $\mathbb{F}$, and for any unifier μ of $\mathbb{F}$, there is a unifier λ such that $\mu = \sigma \circ \lambda$.*

Definition 6. (***Sub-goals***) *Given a fact F, and a rule $R = H \vdash F_1, \ldots, F_n$, where F is unifiable with H with the most general unifier σ, then the resolution $F \circ_{(F,H)} R$ results in sub-goals $F_1\sigma$, ..., $F_n\sigma$.*

$F\sigma$ denotes the application of the unifier σ to the variables inside a fact F. For instance, $\sigma = \{\text{sp} \mapsto EV, \text{bill} \mapsto \theta V, TT \mapsto TV, \text{5y} \mapsto DD\}$, $F = \text{RECEIVEAT}(\theta V, EV, \textbf{Time}(TV))$.

Algorithm 1: **ConformanceCheck**(initgoal, Architecture, Rulesets, N)

Result: Proof found (1) /Proof not found (0)
Inputs: 1. Ruleset; 2. Architecture; 3. *initgoal* (*initgoal* $\in \mathbb{AG}$); 4. Allowed layers of nested crypto functions: N.
if *initgoal* $\in$ *Architecture* ***or*** *can be unified with an element of it* **then**
  return 1
else
  if *initgoal cannot be proved with any rule in Ruleset (i.e. Algorithm 2 returns 0 for all rules)* **then**
    return 0
  else
    return 1
  end
end

Algorithm 2: Prove *initgoal* with a rule, rule $\in$ Ruleset

if *initgoal can be unified with the head of rule, rule* $\in$ *Ruleset, which results in a set of sub-goals, SetofSubgoals (Definition 6)* **then**
  if $\exists$ *subgoal in SetofSubgoals that contains more than N nested layers of crypto functions* ***and*** *rule* $\in$ *{P8, P9}* **then**
    return 0
  else
    if $\exists$ *subgoal in SetofSubgoals that cannot be proved with any rule in Ruleset* ***and*** *is not unifiable with any fact in Architecture* **then**
      return 0
    else
      return 1
    end
  end
else
  return 0
end

The verification goals: In the following, we highlight how the verification goals are generated. Each goal is related to a requirement in the sub-policies in π_θ.

For instance, for the data collection sub-policy $\pi_{\textbf{col}}$ with $cons \in \{Y, N\}$, and the collection purpose values $\{cp_1{:}\theta'_1, \ldots, cp_n{:}\theta'_n\}$, the corresponding verification goals are defined by $\mathcal{G}^\theta_{\textbf{col}} = \mathcal{G}^\theta_{\textbf{ccons}} \cup \mathcal{G}^\theta_{\textbf{cpurp}}$, where $\mathcal{G}^\theta_{\textbf{col}}$ is a set of goals related to $\pi_{\textbf{col}}$, and:

1. $\mathcal{G}^\theta_{\textbf{ccons}} = \{\text{CCONSENTCOLLECTED}(sp, \theta)\}$, the set of verification goals that capture whether *sp* collected the consents for collecting data of type θ.
2. $\mathcal{G}^\theta_{\textbf{cpurp}} = \bigcup_{j=1}^{n}\{\text{CPURPOSE}(\theta'_j, cp_j)\}$, the set of goals that capture the collection purposes, where θ'_j is the resulted data after the service provider applies an action cp_j on a piece of data type θ.

Similarly, the verification goals $\mathcal{G}^\theta_{\textbf{use}}$, $\mathcal{G}^\theta_{\textbf{str}}$, $\mathcal{G}^\theta_{\textbf{del}}$, $\mathcal{G}^\theta_{\textbf{fw}}$, $\mathcal{G}^\theta_{\textbf{has}}$, $\mathcal{G}^\theta_{\textbf{link}}$ for the data usage ($\pi_{\textbf{use}}$), storage ($\pi_{\textbf{str}}$), deletion ($\pi_{\textbf{del}}$), transfer ($\pi_{\textbf{fw}}$), possession ($\pi_{\textbf{has}}$) and linkability ($\pi_{\textbf{link}}$) sub-policies, respectively, can be derived (see Appendix C for more details).

Let us define the following sets that we will use in the automated proof (Algorithm 1), namely:

- *Ruleset* contains all the inference rules in subsection 6.1 (Figures 4-5).
- All the goals generation rules $\mathbb{AG} = \bigcup_{\forall\theta \in DataType_{pol}^{service}} \{\mathcal{G}^\theta_{\textbf{col}} \cup \mathcal{G}^\theta_{\textbf{use}} \cup \mathcal{G}^\theta_{\textbf{str}} \cup \mathcal{G}^\theta_{\textbf{del}} \cup \mathcal{G}^\theta_{\textbf{fw}} \cup \mathcal{G}^\theta_{\textbf{has}} \cup \mathcal{G}^\theta_{\textbf{link}}\}$.
- *Architecture* contains the logical facts that define the architecture, *Architecture* = set of actions $\cup$ *PurpSet* $\cup$ *UniqueTypes*.
  Here, *PurpSet* denotes the set of logic facts that specifies the collection, usage, storage and transfer purposes derived from actions, while *UniqueTypes* specifies the set of facts for unique data types. Specifically, *UniqueTypes* = $\{\text{UNIQUE}(\theta_1), \ldots, \text{UNIQUE}(\theta_n)\}$.

Explanation: As inputs, Algorithm 1 expects the set of inference rules *Ruleset*, a set of facts *Architecture* that captures the actions in an architecture, a verification goal *initgoal*, and N, which is a finite number that denotes the maximum layers of nested cryptographic functions in a piece of data that the algorithm examines.

1. If *initgoal* is an action, then we check whether it can be unified with or equal to a fact in *Architecture*, in which case, 1 is returned, or 0 otherwise.
2. During a proof attempt of either *initgoal* or a *subgoal* with the facts in *Architecture*, we also check if the goal can be unified with purpose-fact (in the set *PurpSet*), or a unique type fact (in *UniqueTypes*).
3. If *initgoal* is not an action fact, purpose-fact, or a unique type, then we check if it can be proved with any inference rule in Ruleset. In case there is no rule that can be used to prove the goal, 1 is returned, otherwise, 0.
4. In Algorithm 2, a step *initgoal* $\circ_{(initgoal,\ head\ of\ rule)}$ *rule* can be successful or unsuccessful (in case there is no unifier σ for *initgoal* and the head of *rule*). This step results in the new sub-goals to be proved. If there is a new sub-goal that contains more than

N layers of nested cryptographic functions (*Senc*, *Aenc*, *Mac*, *Hash*), then we return 0 (i.e. this "branch" of the proof was unsuccessful). A proof can be seen as a derivation tree (as shown in Figure 1), with *initgoal* in the root and the actions/facts in *Architecture* are the leaves. If there is a new sub-goal which corresponds to an architectural action, then we attempt to prove it using the facts in *Architecture* (like in points 1-2 above).

5. The proof process is applied recursively on all the sub-goals resulted from each resolution step, until we run out of sub-goals to be proved.

Two simple examples of how Algorithms 1-2 work can be found in Appendix E. More details about the verification steps can be found in the pre-print report on the GitHub page of the tool [38].

DataProVe supports any layer of nested encryption and will terminate if N is finite. N is defined to achieve termination, completeness, and to make the verification quicker if the user defined actions only contain small number of nested encryptions. This is because our verification engine attempts to search for all possible proof trees (to achieve completeness, in case no proof can be found for a goal). Note that in practice, we can mostly see 1-2 layers of encryption in data. Regarding the representation of encryption, in terms of programming technique, there are several ways to make it effective. A solution we use is storing an encrypted data in a dictionary, where the plaintext is a dictionary key, and the cryptographic keys required to decrypt the plaintext are the dictionary values. This speeds up the search for the resolution/unification steps for decryption.

In the following, we discuss the correctness, termination and completeness properties of our algorithms. In Properties 1-3, $\pi_\theta.\pi_*$ refers to the sub-policy π_* in π_θ, and **ConformanceCheck**(...) refers to **ConformanceCheck**(*initgoal, Architecture, Rulesets, N*).

Property 1. *(Correctness) We distinguish several cases based on the value of initgoal:*

1. *If initgoal* $\in$ *{HAS(E, θ), HASUPTO(E, θ,* ***Time****(dd))}, and* $E \in \pi_\theta.\pi_{has}$ *at the policy level, then whenever* ***ConformanceCheck****(...) == 1, Architecture functionally conforms with* π_θ *(with regards to the requirement* $E \in \pi_\theta.\pi_{has}$*).*
2. *If initgoal* $\in$ *{HAS(E, θ), HASUPTO(E, θ,* ***Time****(dd))}, and* $E \notin \pi_\theta.\pi_{has}$*, then whenever* ***ConformanceCheck****(...) == 1, Architecture does not privacy conform with the policy* π_θ*.*
3. *If initgoal* $\in$ *{LINK(*E_i*,θ,*θ_i*), LINKUNIQUE(*E_i*,θ,*θ_i*)}, and* $(E_i, \theta_i) \in \pi_\theta.\pi_{link}$*, then whenever* ***ConformanceCheck****(...) == 1, Architecture functionally conforms with* π_θ *(with regards to the requirement* $(E_i, \theta_i) \in \pi_\theta.\pi_{link}$*).*
4. *If initgoal* $\in$ *{LINK(*E_i*,θ,*θ_i*), LINKUNIQUE(*E_i*,θ,*θ_i*)} and* $(E_i, \theta_i) \notin \pi_\theta.\pi_{link}$*, then whenever* ***ConformanceCheck****(...) == 1, Architecture does not privacy conform with* π_θ*.*
5. *If initgoal is a consent collection (e.g. initgoal* $\in \mathcal{G}^\theta_{ccons}$*), and* $\pi_{col}.cons = Y$*,* $\pi_{use}.cons = Y$*,* $\pi_{str}.cons = Y$*, or* $\pi_{fw}.cons = Y$ *in* π_θ*, then Architecture DPR conforms with the corresponding sub-policy (e.g.* $\pi_{col}.cons = Y$*) whenever* ***ConformanceCheck****(...) == 1.*

Property 2. *(Termination up-to N) Let N be the maximum number of nested layers of cryptographic functions that Algorithm 2 examines. Also, we assume that all the defined data types contain only finite layers of other data types inside them. Beside a finite N, the proof process never get into an infinite loop.*

The completeness property is a consequence of the termination property (Property 2), as follows:

Property 3. *(Completeness) If all the data types specified in Architecture contain a finite number of layers of cryptographic functions, and all the defined data types have a finite number of layers of other data types, then:*

1. *If initgoal* $\in$ *{HAS(E,θ), HASUPTO(E,θ,Time(dd))}, and* $E \in \pi_\theta.\pi_{has}$ *at the policy level, then whenever* ***ConformanceCheck*** *(...) == 0, Architecture does not functionally conform with* π_θ*.*
2. *If initgoal* $\in$ *{LINK(E,θ,*θ'*), LINKUNIQUE(E,θ,*θ'*)}, and* $(E, \theta') \in \pi_\theta.\pi_{link}$*, then whenever* ***ConformanceCheck****(...) == 0, Architecture does not functionally conform with* π_θ*.*

Similarly, the completeness properties for consent collection can be defined. The proof outline of Properties 1-3 can be found in Appendix D.

7 Implementation and Usage

DataProVe is written in Python, and is available for download (with its user manual) from GitHub [38]. We demonstrate the usage of the tool on a decentralised contact tracing approach (DP3T [39]) and a version of a centralised approach (PEPP-PT [40]), which are

adapted in some European countries. In the following, we define a common data protection policy ($\mathcal{PL}_1$), with the focus of being privacy friendly, but at the same time being effective functionally (e.g. enabling a kind of "global monitoring" of the cases). We will then compare a version of DP3T and PEPP-PT against this policy. To compare the two architectures, $\mathcal{PL}_1$ combines some principles and data types from both the decentralised and centralised designs (such as long-term ID, which normally can only be found in the centralised case).

We define the following **entities** that are involved in the service, (i) the service provider (*sp*), (ii) *phone*, (iii) health authority (*healthauth*), (iv) user (*user*), and (v) backend server (*backend*). We assume that *sp* and *healthauth* are two different entities, while *sp* operates the app and the backend server, *healthauth* deals with testing and notifying users (not via the same app).

The **data types** we define are (i) types of long-term pseudo IDs (*longID*), (ii) ephemeral IDs of an "own" phone (*ephIDown*), (iii) ephemeral IDs of "other" phones (*ephIDother*), (iv) ephemeral IDs sent by the backend server (*ephIDbackend*), (v) the "own" exposure levels (*exposLevelown*), (vi) test results (*testResultown*), (vii) statistics (*statistics*), and (viii) notifications of being at risk (*atRisk*). Hence, the policy is defined by:

$$\mathcal{PL}_1 = \{\pi_{ephIDown}, \pi_{ephIDother}, \pi_{ephIDbackend}, \pi_{longID}, \pi_{exposLevelown}, \pi_{testResultown}, \pi_{statistics}, \pi_{atRisk}\}.$$

For brevity, instead of detailing all the seven sub-policies for each data type in $\mathcal{PL}_1$, in the following, we focus on the most relevant requirements for contact tracing apps (see the tool's GitHub page [38] for the rest).

- For data usage sub-policy in $\pi_{ephIDown}$, *ephIDs* are used to calculate the exposure level, and its usage require prior consent (i.e. $\pi_{use}.upurp = \{calculate{:}exposLevelown\}$, and $\pi_{use}.cons = Y$). As for storage, *ephIDs* are not stored at the service provider's place, hence $\pi_{str}.where = \{decentralised\}$, and consent is required ($\pi_{str}.cons = Y$). *ephIDs* are deleted within 14 days from the users' phone (i.e. $\pi_{del}.fromwhere = \{phone\}$, and $\pi_{del}.deld = 14d$). *ephIDs* are only shared with the (other) phones and backend/service provider, hence $\pi_{fw}.cons = Y$, $\pi_{fw}.fwto = \{phone\}$. We set that the phones, backend/service provider have the right to have *ephIDs* (i.e. $\pi_{has}= \{backend, phone\}$). Finally, only phones have the right to link two of their own *ephIDs* (i.e. $\pi_{link} = \{(phone, ephIDown)\}$).
- For the data storage sub-policy in π_{longID}, to be more privacy-friendly, we set that long-term IDs can only be stored on an own phone (i.e. $\pi_{str}.where = \{phone\}$, and $\pi_{str}.cons = Y$). We set its retention delay to 3 years, the anticipated duration of the pandemic (i.e. $\pi_{del}.fromwhere = \{phone\}$, and $\pi_{del}.deld = 3y$). In addition, we set that only the phones have the right to have the long-term ID (i.e. $\pi_{has}= \{phone\}$), and only the own phones have the right to link *longID* with the ephemeral IDs, the exposure levels, and the test results (i.e. $\pi_{link} = \{(phone, ephIDown)\}$).
- In $\pi_{exposLevelown}$, the exposure level is stored locally on the own phones, and only the users and their own phones have the right to have *exposLevelown*, and link *exposLevelown* with *ephIDs*. Hence, $\pi_{str}.where = \{phone\}$, $\pi_{del}.fromwhere = \{phone\}$, $\pi_{del}.deld = 14d$, $\pi_{has}= \{user, phone\}$, and $\pi_{link} = \{(user, ephID), (phone, ephID)\}$).
- For data possession and connection in $\pi_{testResultown}$, only the user and the health authority have the right to have *testResultown*, and only the user has the right to link its own test result with its own ephemeral ID. Hence, $\pi_{has}= \{user, healthauth\}$, and $\pi_{link} = \{(user, ephID)\}$).
- Finally, in $\pi_{statistics}$, we set that the health authority has the right to have the statistics about the "global cases" (i.e. $\pi_{has}= \{healthauth\}$). In π_{atRisk}, we set that the phones have the right to have the at risk notifications (i.e. $\pi_{has}= \{phone\}$).

In the common policy $\mathcal{PL}_1$, we mapped the requirements in the policy text (white papers) into our seven sub-policy formats. For example, the retention delay in Section 3 (21 days), and consent collection in Section 4 of the PEPP-PT white paper [40]. The data possession policy in $\mathcal{PL}_1$ is from Section 1.2.1 (F-REQ-10 and F-REQ-8). While the linkability property addresses the requirement NF-REQ-8 in Section 1.2.2. Similarly, some requirements are taken from the white paper of DP3T [39], including the 14 days retention delay (page 16 [39]).

7.1 Decentralised Architecture (DP3T)

We consider the low-cost version of DP3T, where each phone generates a list of random ephemeral IDs (EphIDs) from a seed, *SK*, and broadcasts them. A phone stores its own EphIDs, and the ones it heard from other phones for 14 days. For security reason, the people tested positive for COVID-19 have a choice to upload a pair of a seed and the time, (*SK*, *t*), for the contagious period to a backend server, which can be downloaded

by phones to calculate the ephemeral IDs (EphIDs) to assess the exposure level by comparing them with the previously heard EphIDs. As we focus on privacy properties, for simplicity, we assume that EphIDs (instead of the seeds) are uploaded to the backend server and downloaded by phones for calculating the exposure level.

The architecture actions that model the collection of consents can be found in Appendix A (the full list of actions can be found on the GitHub page of the tool).

Architecture: We define the architecture for the low-cost version of DP3T as $\mathcal{PA}_{DP3T}$ that contains:

1. OWN(**user**, ephIDown),
2. OWN(**user**, expoLevelown),
3. OWN(**user**, testResultown),
4. OWN(**phone**, expoLevelown),
5. OWN(**phone**, ephIDown),
6. CALCULATEAT(**phone**, ListEphOwn, **Time**(TT)),
7. CALCULATEAT(**phone**, expoLevelown, **Time**(TT)),
8. RECEIVEAT(**phone**, ephIDother, **Time**(TT)),
9. STOREAT(**phone**, ephIDown, **Time**(TT)),
10. STOREAT(**phone**, ephIDother, **Time**(TT)),
11. STOREAT(**phone**, expoLevelown, **Time**(TT)),
12. DELETEWITHIN(**phone**, ephIDown, **Time**(14d)),
13. DELETEWITHIN(**phone**, ephIDother, **Time**(14d)),
14. DELETEWITHIN(**phone**, expoLevelown, **Time**(14d)),
15. RECEIVEAT(**backend**, ListEphOwn, **Time**(TT)),
16. RECEIVEAT(**backend**, ListEphOther, **Time**(TT)),
17. STOREAT(**mainstorage**, ephIDown, **Time**(TT)),
18. RECEIVEAT(**phone**, ephIDbackend, **Time**(TT)).

7.2 Centralised Architecture (PEPP-PT)

In case of PEPP-PT, the backend server calculates *ephID*s and sends them to the phones, and it also calculates the long-term pseudo ID (*longID*) for each phone/app. The infected users can volunteer to upload the emphemeral IDs (*ephIDs*) their phones heard from other phones. Upon receiving a list of *ephIDs*, the backend server identifies the long-term IDs corresponding to the *ephIDs*, and notify the users at risk as well as the health authority about those users at risk. As stated in Section 4 of [40], *ephIDs* are stored until 21 days on the phones. To show better the ability of our tool and highlight more differences between the two architectures, we also assume an example implementation *IMP* of PEPP-PT, where for the authority to calculate case statistics, *longID* of an infected person is sent to the health authority (but *ephIDs* are not).

Architecture: The architecture of the implementation *IMP* of PEPP-PT is defined as $\mathcal{PA}_{PEP-IMP}$ that contains the following 24 actions:

1. OWN(**user**, ephIDown),
2. OWN(**user**, testResultown),
3. OWN(**backend**, ephIDown),
4. OWN(**backend**, ephIDother),
5. OWN(**backend**, longID),
6. OWN(**backend**, exposLevelown),
7. CALCULATEAT(**backend**, ListEphOwn, **Time**(TT)),
8. CALCULATEAT(**backend**, ListEphOther, **Time**(TT)),
9. CALCULATEAT(**backend**, longID, **Time**(TT)),
10. CALCULATEAT(**backend**, exposLevelown, **Time**(TT)),
11. STOREAT(**mainstorage**, ephIDown, **Time**(TT)),
12. STOREAT(**mainstorage**, longID, **Time**(TT)),
13. STOREAT(**mainstorage**, exposLevelown, **Time**(TT)),
14. DELETEWITHIN(**mainstorage**, longID, **Time**(3y)),
15. RECEIVEAT(**phone**, ephIDown, **Time**(TT)),
16. RECEIVEAT(**phone**, ephIDother, **Time**(TT)),
17. RECEIVEAT(**phone**, atRisk, **Time**(TT)),
18. RECEIVEAT(**backend**, ephIDother, **Time**(TT)),
19. STOREAT(**phone**, ephIDown, **Time**(TT),
20. STOREAT(**phone**, ephIDother, **Time**(TT),
21. DELETEWITHIN(**phone**, ephIDown, **Time**(21d)),
22. DELETEWITHIN(**phone**, ephIDother, **Time**(21d)),
23. RECEIVEAT(**healthauth**, longID, **Time**(TT)),
24. CALCULATEAT(**healthauth**, statistics, **Time**(TT).

ListEphOwn and ListEphOther denote List(ephIDown, ephIDown) and List(ephIDother, ephIDother), respectively, in the architectures.

7.3 Verification Results

Using our tool, DataProVe, the automated verification returns that $\mathcal{PA}_{DP3T}$ violates some *functional properties* against $\mathcal{PL}_1$. Examples of these violations include:

1. The long-term IDs cannot be stored on the phones, as specified in $\mathcal{PL}_1$.
2. The phones cannot have a long-term ID (*longID*).
3. The phones cannot link *longID* with *ephID*.
4. The health authority cannot have the statistics about “global cases" (*statistics*).

Since $\mathcal{PL}_1$ favours the decentralised approach, we can mostly see functional violations for DP3T (e.g. statistics calculation, and requirements on *longID*, which can be found only in PEPP-PT, but also defined in $\mathcal{PL}_1$). DataProVe also returns that $\mathcal{PA}_{DP3T}$ conforms with the privacy properties regarding data possession and linkability (e.g. *backend* cannot link two *ephIDown*-s). Note that points 1-3 can be eliminated with a policy dedicated for DP3T, $\mathcal{PL}_{DP3T}$, in which we remove the type *longID*. As normally, the designers would define dedicated policies with just the data types supported by a service. The combined policy $\mathcal{PL}_1$ is only defined to

demonstrate the ability of DataProVe to compare the two architectures, and detect functional violation.

On the other hand, $\mathcal{PA}_{PEP-IMP}$ also violates $\mathcal{PL}_1$ in several aspects. Examples of these violations include:

1. In $\mathcal{PA}_{PEP-IMP}$, *backend* can link two ephemeral IDs of a phone (*ephIDown*), and it can link *ephIDown* with the exposure level (*exposLevelown*), which are not allowed in $\mathcal{PL}_1$.
2. In $\mathcal{PA}_{PEP-IMP}$, *backend* can have the long-term ID (*longID*) and the exposure level (*exposLevelown*), which is not allowed in $\mathcal{PL}_1$.
3. In $\mathcal{PA}_{PEP-IMP}$, *backend* can link a long-term ID with a ephemeral ID, and with the exposure level of a phone, which are not allowed in $\mathcal{PL}_1$.
4. In $\mathcal{PA}_{PEP-IMP}$, the ephemeral IDs, the long-term IDs, and the exposure levels are stored in the *backend* server, which are not allowed in $\mathcal{PL}_1$.
5. In $\mathcal{PA}_{PEP-IMP}$, *ephIDown* are stored until 21 days on the phone, while only 14 days is allowed in $\mathcal{PL}_1$.

To summarise, from the analysis above, we can see that the examined version of DP3T seems to be stricter regarding the data handled by the backend server, which is privacy friendly. However, from the functionality perspective, the PEPP-PT design seems to enable better cases-monitoring possibilities, as in $\mathcal{PA}_{PEP-IMP}$, the health authority can have *statistics* (actions 23-24).

Finally, in the table below, we provide the time it takes for DataProVe-v0.9.8 to return the results above, besides the system parameter of an Intel i7 9700 CPU, 32 GB RAM, Windows 10 and Python 3.8.

	$\mathcal{PA}_{DP3T}$	$\mathcal{PA}_{PEP-IMP}$
$\mathcal{PL}_1$	**6.432 sec (29 actions)**	**11.605 sec (39 actions)**

We note that the table indicates the time for verifying all the sub-policies requirements defined in $\mathcal{PL}_1$ at once. Each goal took different time between the range of 9ms - 314ms. PEP-IMP takes more time because it has to deal with more intensive verification for a greater number of architecture actions (39 in total).

8 Performance

The verification takes the longest time for the link goals, as all inference rules (for the link property) and all the combinations of the data in the actions are checked. The worst case scenario is when no proof is found. The verification time for a goal increases with the number of architecture actions and arguments in compound data types (that contains other data types as arguments).

Figures 6-7 show the verification time for a single link goal, besides the growing number of actions and arguments in compound data types. In each case, we choose representative sets of the different actions, and the data types with arguments that would generate high number of verification steps in case no proof is found (i.e. all different data types). In Figure 6, the actions do not contain any cryptographic data types, while in Figure 7, each action contains one layer of encryption. The (colored) lines in the figures are distinguished by the number of arguments. For example, the line with the label “5 args" refers to the settings where each action includes a data type that contains 5 data types as arguments (e.g. *Account(name, address, age, job, salary)*), while “0 arg" captures where each action only contains a simple data type (e.g. *name*). We run each settings 10 times, and took their average (with DataProVe-v0.9.8 and an Intel i7 6700 CPU).

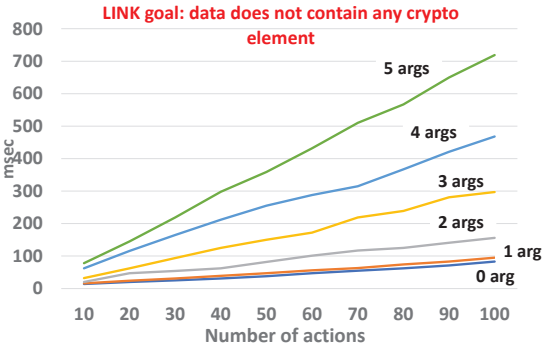

Fig. 6. The verification time of a single LINK goal, with data that does not contain any crypto element. All different data types.

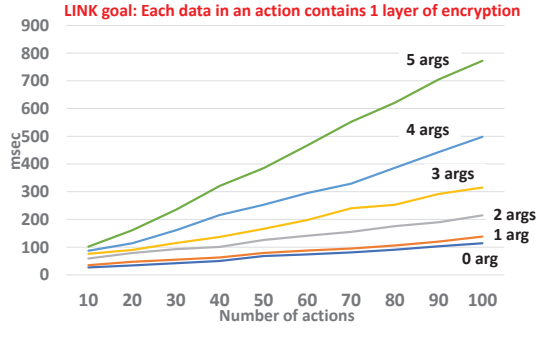

Fig. 7. The verification time of a single LINK goal, with each data containing 1 layer of encryption. All different data types.

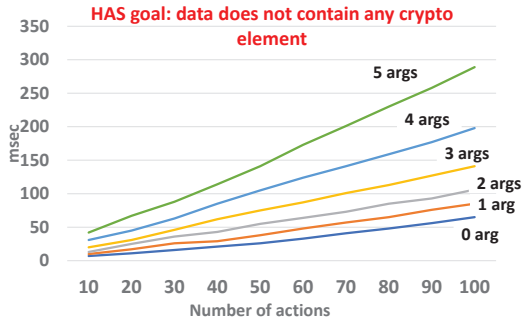

Fig. 8. The verification time of a single HAS goal, with data that does not contain any crypto element. All different data types.

The more arguments a data type has and more actions an architecture has, the more steps are required to find a proof. This can increase the verification time in the worst case scenario when all combinations of argument and action pairs needed to be checked. In Figure 7, the verification time for the encryption cases is slightly higher, than the corresponding non-encryption cases because to access the same data, first, decryption rules need to be invoked. The time difference is small, because rules *P8-9* are still invoked in the non-encryption cases (to check for the existence of encryption in data).

The time of the other goals for consent collection, purposes, deletion delay and storage is much lower, than the link cases as their proofs do not involve a recursive application of the inference rules. The verification time for these is up to *47 msec* besides 100 actions, and 5 arguments. The data possession goal requires recursive application of rules, however, it does not require matching all possible combinations of arguments and data types like the link goals (see Figure 8). The cryptographic elements such as hash/MAC functions do not increase the verification time compared to the non-encryption case, as (being one-way) they do not include any decryption.

9 Future Directions

We discuss some future directions to address the limitations discussed in Section 3.1. To capture the requirements on children consent (*Article 6(1)(f)*), for example, an architecture relation *Role(guardian, E1, E2)* could be defined to specify that *E1* is the guardian of *E2*, and the consent can be received from *E1*. For *Article 6(1)(b)*, an abstract set of interests could be defined. As for Article 9, which deals with the processing of special category data, it could be modelled by defining a set of special category data types (a subset of all data types). Then, based on each condition in *Article 9(2)*, the special purposes of employment, health-check, public health, and research could be defined in the collection and usage sub-policies (π_{col}, π_{use}) of these data types.

There are some possibilities to improve the transfer sub-policy as well, for example, the GDPR covers the case when personal data is transferred to an international organisation (in a third country), and appropriate agreement and arrangement must be done prior to data transfer [37]. This agreement could be specified in the form of a sticky or conditional policy between a service provider and an organisation. Sticky policies are used in PPL [8] to match the expectation of a client and the obligation offered by a service provider.

Regarding the deletion sub-policy, in the GDPR, the data subject also has the right to request a deletion for their collected data. This can be modelled with an event/action that captures the reception of a deletion request (e.g. $recvdelreq(\theta, place, t)$) and a corresponding deletion action within a specified delay. Finally, transparency, another important part of the GDPR as it captures the "right to be informed", can be defined by an action called *notify* happening before the data collection, usage, storage, deletion, and transfer.

10 Conclusion

In this paper, we addressed the problem of formal specification and automated verification of data protection requirements at the policy and architecture levels. To do that, we proposed two variants of policy and architecture languages to specify a set of data protection requirements set in the GDPR. In addition, we proposed DataProVe, a tool based on the syntax of our languages' variants and an automated verification engine to check the conformance between a policy and an architecture.

The notifications about violation and conformance can help system designers adjust their new policies and architectures to meet their objective regarding privacy, data protection and functionality. However, our method can be used for existing real systems as well, as it can model and analyse any architecture that supports the transmission, calculation, storage and deletion of data.

The syntax and semantics of the languages can be extended to specify more complex laws, such as Article 6 and 9 in the GDPR, as well as automated verification between system architectures and implementations, connecting three levels of system design.

11 Acknowledgement

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

References

[1] EU Parliament. General Data Protection Regulation, 2018. https://eur-lex.europa.eu/eli/reg/2016/679/oj.

[2] Karen Kullo. Facebook sued over alleged scanning of private messages. Bloomberg, 2 January 2014. http://www.bloomberg.com/news/articles/2014-01-02/facebook-sued-over-alleged-scanning-of-private-messages.

[3] Samual Gibbs. Belgium takes Facebook to court over privacy breaches and user tracking. The Guardian, 15 June 2015. http://www.theguardian.com/technology/2015/jun/15/belgium-facebook-court-privacy-breaches-ads.

[4] Sean Buckley. Deleting Google Photos won't stop your phone from uploading pictures. Engaget.com, 13 July 2015. http://www.engadget.com/2015/07/13/deleting-google-photos-wont-stop-your-phone-from-uploading-pict/.

[5] K. Granville. Facebook and Cambridge Analytica: What You Need to Know as Fallout Widens. The New York Times, 19 March 2018. https://www.nytimes.com/2018/03/19/technology/facebook-cambridge-analytica-explained.html.

[6] EU Parliament. GDPR, Article 25, 2018. https://eur-lex.europa.eu/eli/reg/2016/679/oj.

[7] EU Parliament. GDPR, Article 6, 2018. https://eur-lex.europa.eu/eli/reg/2016/679/oj.

[8] S Trabelsi, Akram Njeh, Laurent Bussard, and Gregory Neven. Ppl engine: A symmetric architecture for privacy policy handling. *W3C Workshop on Privacy and data usage control*, pages 1–5, 04 2010.

[9] Monir Azraoui, Kaoutar Elkhiyaoui, Melek Önen, Karin Bernsmed, Anderson Santana De Oliveira, and Jakub Sendor. A-ppl: An accountability policy language. In Joaquin Garcia-Alfaro, Jordi Herrera-Joancomartí, Emil Lupu, Joachim Posegga, Alessandro Aldini, Fabio Martinelli, and Neeraj Suri, editors, *Data Privacy Management, Autonomous Spontaneous Security, and Security Assurance*, pages 319–326, Cham, 2015. Springer.

[10] I. Çelebi. Privacy enhanced secure tropos : A privacy modeling language for gdpr compliance. Master Thesis, 2018.

[11] Vinh-Thong Ta, Denis Butin, and Daniel Le Métayer. Formal accountability for biometric surveillance: A case study. In Bettina Berendt, Thomas Engel, Demosthenes Ikonomou, Daniel Le Métayer, and Stefan Schiffner, editors, *Privacy Technologies and Policy*, pages 21–37. Springer, 2016.

[12] Vinh-Thong Ta and Thibaud Antignac. Privacy by design: On the conformance between protocols and architectures. In Frédéric Cuppens, Joaquin Garcia-Alfaro, Nur Zincir Heywood, and Philip W. L. Fong, editors, *Foundations and Practice of Security*, pages 65–81. Springer, 2015.

[13] A. Barth, A. Datta, J.C. Mitchell, and H. Nissenbaum. Privacy and contextual integrity: framework and applications. In *2006 IEEE Symposium on Security and Privacy (S P'06)*, page 15pp., 2006.

[14] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K. Rajamani, Janice Tsai, and Jeannette M. Wing. Bootstrapping privacy compliance in big data systems. In *2014 IEEE Symposium on Security and Privacy*, pages 327–342, 2014.

[15] Mina Deng, Kim Wuyts, Riccardo Scandariato, Bart Preneel, and Wouter Joosen. A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements. *Requirements Engineering Journal*, 16(1):3–32, 2011.

[16] Open policy agent. https://www.openpolicyagent.org/. Accessed: 2021-05-24.

[17] The Platform for Privacy Preferences. P3P, 2012. http://www.w3.org/P3P/.

[18] The Platform for Privacy Preferences (P3P). APPEL 1.0, 2012. http://www.w3.org/TR/2002/WD-P3P-preferences-20020415/.

[19] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Xpref: a preference language for p3p. *Computer Networks*, 48(5):809 – 827, 2005. Web Security.

[20] M. Alshammari and A. Simpson. A model-based approach to support privacy compliance. *Information and Computer Security,*, 26(4):437–453, 2018.

[21] Rainer Hörbe and Walter Hötzendorfer. Privacy by design in federated identity management. In *2015 IEEE Security and Privacy Workshops*, pages 167–174, 2015.

[22] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In Wilhelm Schäfer and Pere Botella, editors, *Software Engineering — ESEC '95*, pages 137–153, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[23] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transaction on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[24] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.

[25] J. Perez, I. Ramos, J. Jaen, P. Letelier, and E. Navarro. Prisma: towards quality, aspect oriented and dynamic software architectures. In *Third International Conference on Quality Software, 2003. Proceedings.*, pages 59–66, 2003.

[26] R. Milner. A calculus of mobile processes, i. *Information and Computation*, 100(1):1 – 40, 1992.

[27] Amelia Bădică and Costin Bădică. Fsp and fltl framework for specification and verification of middle-agents. *Int. J. Appl. Math. Comput. Sci.*, 21(1):9–25, March 2011.

[28] R. B. Franca, J. Bodeveix, M. Filali, J. Rolland, D. Chemouil, and D. Thomas. The aadl behaviour annex – experiments and roadmap. In *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 377–382, 2007.

[29] Thibaud Antignac and Daniel Le Métayer. Privacy architectures: Reasoning about data minimisation and integrity. In Sjouke Mauw and Christian D. Jensen, editors, *Security and Trust Management*, pages 17–32. Springer, 2014.

[30] National Information Technology Development Agency. Nigeria Data Protection Regulation, 2019.

[31] State of California Department of Justice. California Consumer Privacy Act, 2018.

[32] Office of the Privacy Commissioner of Canada. Personal Information Protection and Electronic Documents Act, 2000.
[33] EU Parliament. GDPR, Article 46, 2018. https://eur-lex.europa.eu/eli/reg/2016/679/oj.
[34] EU Parliament. GDPR, Article 5, 2018. https://eur-lex.europa.eu/eli/reg/2016/679/oj.
[35] EU Parliament. GDPR, Article 30, 2018. https://eur-lex.europa.eu/eli/reg/2016/679/oj.
[36] EU Parliament. GDPR, Article 17, 2018. https://eur-lex.europa.eu/eli/reg/2016/679/oj.
[37] EU Parliament. GDPR, Article 45, 2018. https://eur-lex.europa.eu/eli/reg/2016/679/oj.
[38] Dataprove. https://github.com/Dataprove/Dataprovetool/.
[39] Carmela Troncoso et. al. Decentralized Privacy-Preserving Proximity Tracing. GitHub, 25 May 2020. https://github.com/DP-3T/documents.
[40] Pan European Privacy Preserving Proximity Tracing. Data Protection and Information System Architecture. GitHub, 20 April 2020. https://github.com/pepp-pt/pepp-pt-documentation.

A Actions for Consent

In the following, we define the actions for consent collections. We use the notations in the table below:

Notation	Data = (θV, $EV_{consent}$)
EphOwn	(ephIDown, phone)
EphOther	(ephIDother, phone)
EphOwnSp	(ephIDown, backend)
EphOtherSp	(ephIDother, backend)
LongIDSp	(longID, healthauth)
EphBackend	(ephIDbackend, phone)
EphOwnSp	(ephIDown, mainstorage)
ExpoStorage	(exposLevelown, mainstorage)

For $\mathcal{PA}_{DP3T}$, we define the following actions:
1. RECEIVEAT(**sp**, **Uconsent**(EphOther), **Time**(TT)) defines the collection of a usage consent on *ephIDother*.
2. RECEIVEAT(**sp**, **Uconsent**(EphBackend), **Time**(TT)) is the same as the first one but on EphBackend.
3. RECEIVEAT(**sp**, **Fwconsent**(EphOther), **Time**(TT)) deals the consent for sharing *ephIDother* with *phone*.
4. RECEIVEAT(**sp**, **Sconsent**(EphOwn), **Time**(TT)) deals with the consent for storing *ephIDown* on *phone*.
5. RECEIVEAT(**sp**, **Sconsent**(EphOther), **Time**(TT)) captures the consent for storing *ephIDother* on *phone*.
6. RECEIVEAT(**sp**, **Sconsent**(ExpoStorage), **Time**(TT)) deals with the storage consent of the exposure level.
7. RECEIVEAT(**sp**, **Fwconsent**(EphOwnSp), **Time**(TT)) deals with the consent of sharing *ephIDown* with *backend*.
8. RECEIVEAT(**sp**, **Sconsent**(EphOwnSp), **Time**(TT)) deals with the storage consent for *ephIDown* at backend.

Similarly, for $\mathcal{PA}_{PEP-IMP}$, we define the actions:
1. RECEIVEAT(**sp**, **Uconsent**(EphOwnSp), **Time**(TT)),
2. RECEIVEAT(**sp**, **Uconsent**(EphOtherSp), **Time**(TT)),
3. RECEIVEAT(**sp**, **Uconsent**(LongIDSp), **Time**(TT)),
4. RECEIVEAT(**sp**, **Fwconsent**(EphOther), **Time**(TT)),
5. RECEIVEAT(**sp**, **Fwconsent**(LongIDSp), **Time**(TT)),
6. RECEIVEAT(**sp**, **Sconsent**(EphOwn), **Time**(TT)),
7. RECEIVEAT(**sp**, **Sconsent**(EphOther), **Time**(TT)),
8. RECEIVEAT(**sp**, **Sconsent**(EphOwnSp), **Time**(TT)),
9. RECEIVEAT(**sp**, **Sconsent**(ExpoStorage), **Time**(TT)).

B Cryptographic and Attacker Models

In the following, we provide an abstract model for homomorphic encryption. We define the function **HomEnc**(**Sum**(X_{num1}, X_{num2},..., X_{numn}), X_{key}), where **HomEnc** denotes homomorphic encryption, and **Sum** models summation, but other operation can also be defined in place of it. We also define the action

CALCULATEFROMAT(E, $X_{\theta 1}$, $X_{\theta 2}$, **Time**(TT)),

which for brevity, we did not include in Fig. 2. This specifies that E can calculate $X_{\theta 1}$ from $X_{\theta 2}$ at time TT. Then, for **HomEnc**, we define n corresponding actions:
– CALCULATEFROMAT(E, X_θ, $X_{\theta 1}$,**Time**(TT_1)),
...,
– CALCULATEFROMAT(E, X_θ, $X_{\theta n}$,**Time**(TT_n))),
where X_θ = **HomEnc**(**Sum**(X_{num1}, X_{num2},..., X_{numn}), X_{key}), and $X_{\theta 1}$ = **Enc**(X_{num1}, X_{key}),..., $X_{\theta n}$ = **Enc**(X_{numn}, X_{key}).

The decryption for **HomEnc** is defined based on a destructor similar to the asymmetric decryption.

B.1 Our Attacker Model

We consider three types of attackers. First, the external attackers can only eavesdrop on the communication if a message can be received on public channels (modelled by the RECEIVEAT action). Encryption can be used if the designer wants to "hide" the content from the attacker. Eavesdropping can be formalised/modelled with the following inference rule:

HAS(att, θV) $\vdash$ RECEIVEAT(EV, θV, **Time**(TV)), where *att* denotes the attacker entity.

Specifically, if EV can receive θV at some non-specific time TV via a public channel, then the attacker

can have θV. The external attackers can decrypt, possess or link the received data if it can have the key, or apply one the HAS or LINK inference rules on the data.

The insider attackers are compromised entities, and therefore, they can only receive, possess or link the data that the entity can. The case of collusion among insider attackers is more interesting as the attackers can share the data of the compromised entities with each other. This can be formalised by specifying that the attacker has access to the compromised entities (with the construct $HasAccessTo(att) = \{E1,\ldots, En\}$). Finally, the hybrid case is the most powerful attacker model, as it assumes the collusion (data shared) between external and insider attackers. Its formalisation is the combination of the formalisations of the previous two cases.

C Verification Goals Generation

We provide the other rules for the generation of verification goals.

1. For $\pi_{\mathbf{use}}$ with $cons \in \{Y, N\}$, and the usage purpose values $\{up_1{:}\theta'_1,\ldots, up_n{:}\theta'_n\}$, the following verification goals are generated:

$\mathcal{G}^{\theta}_{\mathbf{use}} = \mathcal{G}^{\theta}_{\mathbf{ucons}} \cup \mathcal{G}^{\theta}_{\mathbf{upurp}}$, where $\mathcal{G}^{\theta}_{\mathbf{ucons}} = \{$UCONSENTCOLLECTED($sp$, θ)$\}$, and

$\mathcal{G}^{\theta}_{\mathbf{upurp}} = \bigcup_{j=1}^{n}\{$UPURPOSE($\theta'_j$, up_j)$\}$.

2. For $\pi_{\mathbf{str}}$ with $cons \in \{Y, N\}$, and the storage place values $\{E_1,\ldots, E_n\}$, the next verification goals are generated:

$\mathcal{G}^{\theta}_{\mathbf{str}} = \mathcal{G}^{\theta}_{\mathbf{scons}} \cup \mathcal{G}^{\theta}_{\mathbf{places}}$, where

$\mathcal{G}^{\theta}_{\mathbf{scons}} = \{$STRCONSENTCOLLECTED($sp$, θ)$\}$,

$\mathcal{G}^{\theta}_{\mathbf{places}} = \bigcup_{j=1}^{n}\{$STORE($E_j$, θ)$\} \cup \bigcup_{j=1}^{n}\{$STOREAT($E_j$, θ, $\mathbf{Time}(TT)$)$\}$.

Note that E_j/E denotes an entity value (e.g. E = phone).

3. If $\pi_{\mathbf{del}} = (\{E_1,\ldots, E_n\}, dd)$, where $E_1,\ldots, E_n$ are the values of the deletion places, and dd is the value of the deletion delay, then:

$\mathcal{G}^{\theta}_{\mathbf{del}} = \mathcal{G}^{\theta}_{\mathbf{hasupto}} \cup \mathcal{G}^{\theta}_{\mathbf{within}}$, where $\mathcal{G}^{\theta}_{\mathbf{hasupto}} = \bigcup_{j=1}^{n}\{$HASUPTO($E_j$, θ, $\mathbf{Time}(dd)$)$\}$,

$\mathcal{G}^{\theta}_{\mathbf{within}} = \bigcup_{j=1}^{n}\{$DELETEWITHIN($E_j$, θ, $\mathbf{Time}(dd)$)$\}$.

4. If $\pi_{\mathbf{fw}} = (cons, \{E_1,\ldots, E_n\}, \{fwp_1{:}\theta'_1,\ldots, fwp_m{:}\theta'_m\})$, where $E_1,\ldots, E_n$ are the entities who can receive the transferred data, and $fwp_1,\ldots, fwp_m$ are the transfer purpose values, then:

$\mathcal{G}^{\theta}_{\mathbf{fw}} = \mathcal{G}^{\theta}_{\mathbf{fwcons}} \cup \mathcal{G}^{\theta}_{\mathbf{fwto}} \cup \mathcal{G}^{\theta}_{\mathbf{fwpurp}}$, where

$\mathcal{G}^{\theta}_{\mathbf{fwto}} = \bigcup_{j=1}^{n}\{$RECEIVE($E_j$, θ), RECEIVEAT(E_j, θ, $\mathbf{Time}(TT)$)$\}$

$\mathcal{G}^{\theta}_{\mathbf{fwcons}} = \bigcup_{j=1}^{n}\{$FWCONSENTCOLLECTED(sp, θ, E_j)$\}$,

$\mathcal{G}^{\theta}_{\mathbf{fwpurp}} = \bigcup_{j=1}^{m}\{$FWPURPOSE($E_j$, θ'_j, fwp_j)$\}$.

5. For $\pi_{\mathbf{has}}$, if $\{E_1,\ldots, E_n\}$ is a set of all defined entities in an architecture, then:

$\mathcal{G}^{\theta}_{\mathbf{has}} = \bigcup_{j=1}^{n}\{$HAS($E_j$, θ)$\}$.

6. For $\pi_{\mathbf{link}}$, if $\{E_1,\ldots, E_n\}$ is a set of all the defined entities in an architecture, and $\{\theta_1,\ldots, \theta_m\}$ is the set of all defined data types (different from θ), then:

$\mathcal{G}^{\theta}_{\mathbf{link}} = \bigcup_{i=1,j=1}^{n,m}\{$LINK($E_i$, θ, θ_j)$\} \cup$
$\bigcup_{i=1,j=1}^{n,m}\{$LINKUNIQUE($E_i$, θ, θ_j)$\}$.

D Proofs

D.1 Proof of Property 1

Proof. **ConformanceCheck**(*initgoal, Architecture, Rulesets, N*) == 1 means that a proof of *initgoal* can be found with *Architecture*. Whenever *initgoal* can be proved with a rule $rule = H \vdash T_1,\ldots,T_n$ (in Algorithm 2), there is at least one fact in *Architecture* that can be used to prove the sub-goals $T_1,\ldots, T_n$. Besides, since *Data* includes the entity who is given consent for carry out a given action, i.e. $Data = (\theta V, EV_{consent})$, we can avoid that in the rules for consent collections (e.g. D1-D2), the consent contains a different data from the one that can be received by $EV_{consent}$.

Therefore, in case of points 1 and 3 (of Property 1), the first two points of Definition 3 are satisfied, respectively. In case of points 2 and 4, the two points of Definition 1 are unsatisfied, respectively. In case of point 5, **ConformanceCheck**(*initgoal, Architecture, Rulesets, N*) == 1 means that the first point of Definition 2 is satisfied. We refer the readers to the tool's GitHub page for the rest points of Definition 3. □

D.2 Proof of Property 2

Proof. We will show that the number of resolution steps is always finite in the proof of *initgoal*. As a result of a resolution **goal** $\circ_{(goal,\ head\ of\ rule)}$ **rule**, where $rule \in \{P8, P9\}$, we get the two new (sub-)goals in the tails of the rules (e.g. $goal \circ P8 =$ HAS(EV, $\mathbf{Senc}(\theta V,K))\sigma$, HAS($EV$, $K)\sigma$). Since Algorithm 2 does not prove any goal with more than N layers of cryptographic functions (e.g. HAS(sp,$\mathbf{Senc}$($\mathbf{Senc}$,...($\mathbf{Mac}$(name, key))), ...,key),key), there are maximum N recursive calls of

the resolution step **goal** $\circ_{(goal,\ head\ of\ rule)}$ *rule*, where *rule* $\in$ {*P8*, *P9*}. Each recursive call produces two sub-goals, hence, N recursive calls result in at most 2^N sub-goals to be proved. In the worst case, this would mean 2^N*|*Ruleset*| resolution steps (between each goal and rule, where |*Ruleset*| is the number of rules in *Ruleset*).

In case *rule* is one of *P3-P7*, a step *goal* $\circ_{(goal,\ head\ of\ rule)}$ *rule* would generate a single goal (e.g. *goal* $\circ_{(goal,\ head\ of\ P4)}$ *P4* = RECEIVEAT(*EV*, θV, **Time**(*TT*))σ). Then, the resulted (sub-)goals will be checked against the facts in *Architecture*, which yields |*Architecture*| + 1 resolution steps for each rule (where |*Architecture*| is the number of facts in *Architecture*).

In case *rule* is one of *D1-D5* or *P1*, 2*|*Architecture*| + 1 resolution steps are carried out. For *P2*, a step *goal* $\circ_{(goal,\ head\ of\ P2)}$ *P2* generates a single (sub-)goal. The (sub-)goals are then be checked against the rule set (*Ruleset*), including rules *P3-P7*, which yields 2*|*Architecture*| + 1 resolution steps in each case. In addition, when these (sub-)goals are checked against *P8-P10*, it yields 2^N*|*Ruleset*| resolution steps in each case. By invoking *P2*, we cannot have an infinite number of recursive resolution steps between these rules and the resulted sub-goals, because *ds* cannot be unified with *P(ds)*, and θV cannot be unified with either *ds* or *P(ds)*, as being of different types.

In case *rule* is a LINK or LINKUNIQUE rule (Figure 5), a resolution step **goal** $\circ_{(goal,\ head\ of\ rule)}$ **rule** generates two (sub-)goals. Each (sub-)goal will be examined against every rule (in *Ruleset*), but a resolution step can only be successful in case of *P3-P9*. The resolution with each of these rules results in a finite number of further resolution steps (as we argued above). □

D.3 Proof of Property 3

In order to prove Property 3, we need to show that whenever **ConformanceCheck**(...) == 0:

1. Algorithm 1-2 checked all the possible proof branches (if we imagine the proof process as a tree like in Figure 1).
2. The inference rules can capture all possible data types (including the compound data types that contain other data types) can be defined by the user.

The first point is achieved by attempting to prove *initgoal* with all the rules in the inference rule set. We try to unify *initgoal* with the head of the first rule in the rule set, then we attempt to prove the resulted sub-goals with the rule set and architecture, recursively. We also take into account the fact that there can be several proof trees for *initgoal*, and we explore all the possible proof trees. For the second point, we define the inference rules that the Algorithms 1-2 use besides the basic rules in Section 6.1 for the completeness property. Rule D6 is similar to D1, but include the data type Anytypeinccrypto[θV] to deal with complex data that contains other data inside it. E.g. Anytypeinccrypto[θV] can be *Sicknessrec*(θV,...); *Sicknessrec*(Anytypeinccrypto1[θV],...); **Senc**(θV, *K*); or **Senc**(Anytypeinccrypto1[θV], *K*).

D6. FWCONSENTCOLLECTED(*EV*, θV, EV_{to}) ⊢
RECEIVEAT(*EV*, **Fwconsent**(θV, EV_{to}),**Time**(*TV*)),
RECEIVEAT(EV_{to}, Anytypeinccrypto[θV], **Time**(*TV*)).

L0/b. LINK(*EV*, θV_1, θV_2) ⊢
HAS(*EV*, Anytype1(θV_1, θV, **Meta**(θV_3))),
HAS(*EV*, Anytype2([θV_2], $\theta V'$, **Meta**([θV_3])))

L1/c. LINK(*EV*, θV_1, θV_2) ⊢
HAS(*EV*, Anytype1([θV_1], θV, [θV_3])),
HAS(*EV*, Anytype2(θV_2, $\theta V'$, θV_3))

U1/b. LINKUNIQUE(*EV*, θV_1, θV_2) ⊢
HAS(*EV*, Anytype1([θV_1], θV, [θV_3])),
HAS(*EV*, Anytype2(θV_2, $\theta V'$, θV_3)), UNIQUE(θV_3)

P1/b. HASUPTO(*EV*, θV, **Time**(*DD*)) ⊢
STOREAT(*EV*, [θV], **Time**(*TV*)),
DELETEWITHIN(*EV*, [θV], **Time**(*DD*)).

P3/b. HAS(*EV*, θV) ⊢ OWN(*EV*, [θV]).

P8/b. HAS(*EV*, θV) ⊢ HAS(*EV*, **Senc**([θV], *K*)),
HAS(*EV*, *K*).

Fig. 9. The generic version of the basic rules in Section 6.1, which include [θV] instead of θV. [θV] can be either a data type that contains θV inside it, or θV itself.

E Proof Examples (Alg.1-2.)

Example 1. Let *Architecture* = {RECEIVEAT(*sp*, *name*, **Time**(*TT*))} and *initgoal* = HAS(*sp*, *name*), namely, we want to prove that *sp* can have *name*. This can be proven with rule *P4* in Section 6.1 and a resolution step in Definition 6.

– **Step 1**: *initgoal* $\circ_{(initgoal,\ HAS(EV,\theta V))}$ *P4* = RECEIVEAT(*sp*, *name*, **Time**(*TT*))), as *initgoal* can be unified with HAS(*EV*, θV), the head of rule

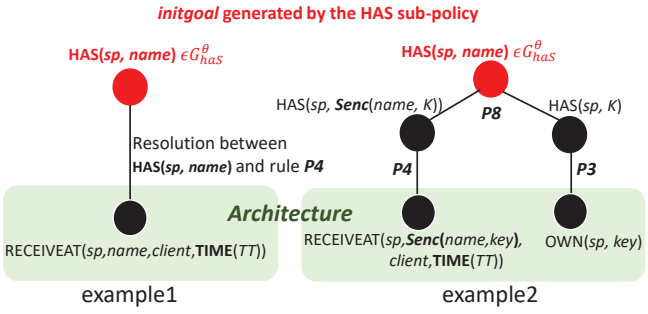

Fig. 10. Two example proofs (without and with encryption).

P4, with the unifier $\sigma = \{sp \mapsto EV,\ name \mapsto \theta V,\ TT \mapsto TV\}$. We have RECEIVEAT($EV$, θV, **Time**(TV))σ as a result, which is equal to RECEIVEAT(sp, $name$, **Time**(TT))).

– **Step 2**: As RECEIVEAT(sp, $name$, **Time**(TT))) $\in$ *Architecture*, therefore, we get **ConformanceCheck**(*initgoal, Architecture, Rulesets, N*) == 1, for any natural N.

Example 2. Let *Architecture* = {RECEIVEAT(sp, **Senc**($name$,key), **Time**(TT)), OWN(sp, key)} and *initgoal* = HAS(sp, $name$). This can be proven with rules *P8*, then *P3*, *P4* as shown in Figure 10.

F Proposed Policy Semantics

The semantics of the policy syntax is based on the events that capture the actions performed by entities during an instance of a system run. An event is defined by a tuple starting with an event name that denotes an action carried out by an entity, followed by the time of the event, and some further action-specific parameters.

Our language supports the events *cconsentat*, *collectat*, *uconsentat*, *sconsentat*, *service_spec_use_event*, *storeat*, *deleteat*, *fwconsentat*, *forwardat*.

To verify the compliance with a policy, we define event trace (τ) that captures a sequence of events happening during a system run. Therefore, the semantics of the policy is defined based on a set of compliant event traces. A compliant event trace contains a sequence of events that complies with the defined sub-policies. For example, if consent is set to "required" in π_{col} ($\pi_{col}.cons = Y$), then an event trace, in which the event *collectat* is preceded by a corresponding event *cconsentat*, can be seen as compliant. Events can have parameters, for example, as follows:

Ev1: (***cconsentat***, t, $E_{consent}$, θ). This event specifies that a data collection consent is being collected at time t by the service provider (sp), in which $E_{consent}$ is given consent to do some actions (e.g. receive, calculate, create) on θ.

Ev2: (***collectat***, t, θ, v) specifies when a piece of data of type θ and value v is collected by sp at time t.

Ev3: (***uconsentat***, t, $E_{consent}$, θ) specifies that a data usage consent is collected by sp at time t on $E_{consent}$ and θ. E.g. *(**uconsentat**, 2020.01.21.11:18, server, gas)*.

Ev4: (***storeat***, t, θ, v, *place*) specifies that a piece of data of type θ and value v is stored at a place *place* at time t.

Ev5: (***deleteat***, t, θ, v, *place*) specifies that at some time t, sp deletes a piece of data of type θ and value v from *place*.

Ev6: (***fwconsentat***, t, E_{to}, θ) specifies that sp is collecting a data transfer consent on a piece of data of type θ.

G Architecture Semantics

Similar to the policies, the semantics of an architecture is based on events and system run traces. A trace Γ is a sequence of high-level events $Seq(\epsilon)$ taking place in during a service, as presented in Figure 11.

$\Gamma ::= Seq(\epsilon)$
$\epsilon ::= own(E, X_\theta{:}V_\theta, \forall t)$, for all t in any traces of a service
| $calculateat(E, X_\theta{:}T, t)$
| $createat(E, X_\theta{:}T, t)$
| $receiveat(E, X_\theta{:}V_\theta, t)$
| $receiveat(E, \textbf{Cconsent}(Data){:}V_{cconsent}, t)$
| $receiveat(E, \textbf{Uconsent}(Data){:}V_{uconsent}, t)$
| $receiveat(E, \textbf{Sconsent}(Data){:}V_{sconsent}, t)$
| $receiveat(E, \textbf{Fwconsent}(Data){:}V_{fwconsent}, t)$
| $storeat(E, X_\theta{:}V_\theta, t)$
| $deletewithin(E, X_\theta{:}V_\theta, dd, t)$.
Where $Data = (X_\theta, E_{consent})$.

Fig. 11. Events defined for architectures.

An event can be seen as an instance of an action defined in Figure 2 that happens at some specific time t (e.g. *2020.01.30.15:45*) during a system run trace. Events are given the same names as the corresponding actions, but in lower-case to avoid confusion. An event specifies that during a system run, at some time t, a piece of data X_θ in an action takes a value V. For example, event $calculateat(E, X_\theta{:}T, t)$ captures that at some time t, E calculates a piece of data of type θ that is equal to a term T (based on the equation $X_\theta{=}T$, e.g. $X_{hash} = Hash(X_{password})$). Event $receiveat(E, X_\theta{:}V_\theta, t)$ says that E receives data X_θ of value V_θ at time t.