Constructing Independently Verifiable Privacy-Compliant Type Systems for Message Passing between Black-Box Components

Citation for the original published paper (version of record):

Adams, R., Schupp, S. (2018)
Constructing Independently Verifiable Privacy-Compliant Type Systems for Message Passing between Black-Box Components
Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and

Constructing Independently Verifiable Privacy-Compliant Type Systems for Message Passing between Black-Box Components

Robin Adams[1] and Sibylle Schupp[2]

[1] Chalmers University of Technology
robinad@chalmers.se
[2] Technische Universität Hamburg-Harburg
sibylle.schupp@tuhh.de

Abstract. *Privacy by design* (PbD) is the principle that privacy should be considered at every stage of the software engineering process. It is increasingly both viewed as best practice and required by law. It is therefore desirable to have formal methods that provide guarantees that certain privacy-relevant properties hold. We propose an approach that can be used to design a privacy-compliant architecture without needing to know the source code or internal structure of any individual component. We model an architecture as a set of *agents* or *components* that pass *messages* to each other. We present in this paper algorithms that take as input an architecture and a set of privacy constraints, and output an extension of the original architecture that satisfies the privacy constraints.

1 Introduction

Privacy by Design is the principle that privacy should be a consideration at every stage of the software design process[8]. It is increasingly seen as best practice for privacy protection, including by the International Conference of Data Protection and Privacy Commissioners[9] and the US Federal Trade Commission[14], and is a legal requirement in the EU since the General Data Protection Regulation (GDPR) came into force on 25 May 2018[13].

It is therefore desirable to create methods that will provide a guarantee that software satisfies certain privacy-relevant properties. To this end, a substantial amount of research (both formal methods and other approaches) has been devoted to this problem, including static analysis of source code (e.g. [15],[11]); real-time "taint tracking" of the data released by apps on a mobile device (e.g. [19],[12]); refinement techniques that preserve privacy properties as we refine in stages from a high-level design to code (e.g. [1],[10]); or the creation of new programming languages which include representations of privacy-relevant properties in types or annotations (e.g. [18],[16]).

We can thus design a privacy-safe application, or verify that a given application is privacy-safe, provided we can access and/or change its source code. However, in practice, many systems involve the interaction of different components, each controlled by a different person or organisation. The source code might not be available, or it might not be possible for us to change it. New versions of each component may come out regularly, so that a privacy analysis we did using an old component quickly becomes obsolete.

In this paper, we will show how we can design a type system for the *messages* that the components pass to each other, in such a way that we can formally prove that, if every message passed is typable under this typing system, then the privacy property must hold. We indicate how an existing unsafe component can be adapted into a component that uses this typing system by providing each component with an *interface* through which all messages must pass, without needing to read or modify the component's source code.

The structure of the paper is as follows. In Section 2, we give a relatively simple but realistic example of privacy constraints that we may wish to hold, and show the architecture that our algorithms generate. In Section 3, we provide the formal definition of architecture that we use. In Section 4, we define the algorithm for a simple constraint language and prove it correct. In Section 5, we do the same development again for a stronger language of constraints, of the form $\alpha \ni A \Rightarrow \beta \ni B$ ('if α possesses a term of type A then β must previously have possessed a term of type B'). Finally we survey some related work in Section 6, and conclude in Section 7.

2 Motivating Example

We now give an example of realistic privacy constraints that we might wish to introduce, and the architectures that are produced by our algorithms. The example is similar to an example considered by Barth et al. [4].

The US Children's Online Privacy Protection Act (COPPA) includes the clause:

> When a child sends protected information to the website, a *parent* must have previously received a privacy notice from the web site operator, [and] granted consent to the web site operator.

We propose to model a system as being composed of *agents* or *components* who pass *messages* to each other. The possible messages are provided by a *type system*, which consists of a set of *types* and a set of *constructors*. These two sets determine the set of *terms*, each of which has a type. We write $t : A$ to denote that the term t has type A.

A *message* is a triple (α, t, β), where α and β are agents and t is a term; this represents the agent α sending the piece of data t to β. If $t : A$, then we write this message as $\alpha \xrightarrow{t} \beta$ or $\alpha \xrightarrow{t:A} \beta$.

For the COPPA example, Figure 1 suggests an architecture with three agents, Child, Website, and Parent. In the initial state, Child possesses a term info : INFO, Website possesses policy : POLICY, and Child may send messages of type INFO to Website, etc. This represents a website which can send its privacy policy to the parent; the parent may send consent for the website to collect the child's protected info; and the child may send their protected info to the website. However, at the moment, there is nothing to prevent the protected info being sent to the website without either policy or consent having been sent.

Formally, an *architecture* is described by specifying the following (see Definition 2):

– for any agent α, which constructors an agent possesses in the initial state;
– for any two agents α, β, the set of types A such that α may pass a message of type A to β.

If A is a type, we shall sometimes say 'α can send A to β' to mean 'α may send messages of type A to β'.

We envision the designer beginning with a set `Ag` of agents and a type system $\mathcal{T}$ which describes the pieces of data they are interested in. They write down the set $\mathcal{C}$ of privacy constraints that they wish the finished system to have. For now, we consider constraints of these two forms (see Definitions 4 and 6):

– $\alpha \ni A \Rightarrow B$: If agent α has a piece of data of type A, then a piece of data of type B must have previously been created.
– $\alpha \ni A \Rightarrow \beta \ni B$: If agent α has a piece of data of type A, then agent β must previously have had a piece of data of type B.

The privacy constraints that we require for the architecture in Figure 1 include

$$Website \ni INFO \Rightarrow Website \ni CONSENT$$

$$Website \ni CONSENT \Rightarrow Parent \ni POLICY$$

The first constraint specifies that agent Website possesses INFO only if it previously has received data of type CONSENT. The second constraint specifies that agent Website possesses CONSENT only if the Parent agent has received the POLICY before. (We will add a third constraint later, in Section 5.1.)

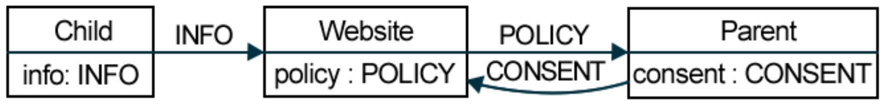

Fig. 1. An Architecture That Allows Privacy Breach

Given privacy constraints, we show how to extend $\mathcal{T}$ to a type system $\mathcal{T}_{\mathbb{C}}$. The type system $\mathcal{T}_{\mathbb{C}}$ includes a set of new types $C_\alpha(A)$. A term of type $C_\alpha(A)$ is called a *certified* term. As well as the plain INFO type, for example, the safe architecture contains the type $C_{Website}(INFO)$. A term of this type represents a piece of data from which $Website$ can extract a term of type $INFO$, but no other agent can.[3] There are no restrictions on which agents may receive them or send certified terms.

The type system $\mathcal{T}_{\mathbb{C}}$ also has types $P_\alpha(A)$, and constructors p_α that construct terms of type $P_\alpha(A)$. We may think of a term of type $P_\alpha(A)$ as a *proof* that α possesses a term of type A.

The architecture created by our Algorithm 2 is shown in Fig. 2. (For space reasons, we have listed only some of the constructors and messages, and omitted the subscripts on the types $C_\alpha(A)$ and $P_\alpha(A)$.) The algorithm creates new components *IWebsite*, the *input interface* to *Website*, and *OWebsite*, the *output interface* for *Website*; and similarly input and output interfaces for *Parent* and *Child*.

The constructor p_{POLICY} takes a term of type $POLICY$ and constructs a term of type $P_{Parent}(POLICY)$ — a proof that *Parent* has received a term of type $POLICY$. The constructor $m_{CONSENT}$ constructs a certified term of type $C_{Website}(CONSENT)$ out of a term of type $CONSENT$, plus the proof that the preconditions for $Website$ to be allowed to read a term of type $CONSENT$, namely a term of type $P_{Parent}(POLICY)$. The constructor $\pi_{CONSENT}$ then extracts the term of type $CONSENT$ from the certified term. Similar comments hold for m_{POLICY} and π_{POLICY}, and the other new constructors in Fig. 2.

It can be seen that, while *Child* may send *INFO* to *OChild* at any time, the only way for the data to travel any further is for a term of type $C_{Website}(INFO)$ to be created; this can only happen if a term of type $P_{Website}(CONSENT)$ has been created; this can only happen if a term of type *CONSENT* reaches *OWebsite*; and this can only happen if *Website* has a term of type *CONSENT*. Similar considerations hold for our other negative constraint.

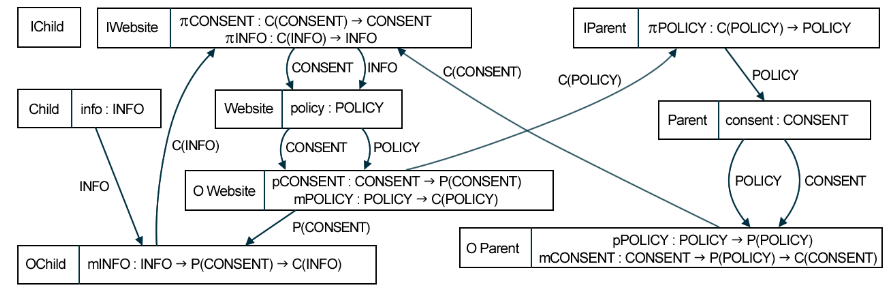
IChild
IWebsite
πCONSENT : C(CONSENT) → CONSENT
πINFO : C(INFO) → INFO
IParent
πPOLICY : C(POLICY) → POLICY
CONSENT
INFO
C(CONSENT)
C(POLICY)
POLICY
Child
info : INFO
Website
policy : POLICY
Parent
consent : CONSENT
C(INFO)
CONSENT
POLICY
INFO
O Website
pCONSENT : CONSENT → P(CONSENT)
mPOLICY : POLICY → C(POLICY)
POLICY
CONSENT
P(CONSENT)
O Parent
pPOLICY : POLICY → P(POLICY)
mCONSENT : CONSENT → P(POLICY) → C(CONSENT)
OChild
mINFO : INFO → P(CONSENT) → C(INFO)

Fig. 2. A Privacy-Safe Architecture

We can partition the agents in Fig. 2 into three sets: $\{Child, IChild, OChild\}$, $\{Website, IWebsite, OWebsite\}$, $\{Parent, IParent, OParent\}$. Each set thus consists of one of the agents from Fig. 1, plus its two new interfaces. Note that, if an agent from one set passes a message to an agent in another set, then that

[3] In practice, this would presumably be achieved by encryption, but we abstract from these implementation details here. See Section 2.1 for more discussion.

message has type $C_\alpha(A)$ or $P_\alpha(A)$ for some α, A. In the rest of this paper, we will prove two results (Theorems 1 and 2) that give general conditions such that, if an architecture can be partitioned in a way that satisfies these conditions, then a given set of privacy constraints are satisfied.

2.1 Note on Implementation

In practice, certification on the one hand, access on the other hand, could be implemented through encryption and decryption. But, other mechanisms possibly exist as well. The type systems we present in this paper abstract from these details. They specify which agents may and may not access which data, without specifying how this is to be done.

The terms of type $P_\alpha(A)$ should, in practice, ideally be an appropriate zero-knowledge proof which guarantees that α possesses a term of type A, without revealing the value of the term of type A. Again, in this paper we abstract from the details of how this would be implemented.

However, we expect it to be possible to implement these types in such a way that the designer could publish both the set of constraints $\mathbb{C}$ and the type system $\mathcal{T}_\mathbb{C}$, and an independent third party (the user, a regulatory authority, or anyone else) to verify both that our algorithm maps $\mathbb{C}$ to $\mathcal{T}_\mathbb{C}$, and that any given message is typable under $\mathcal{T}_\mathbb{C}$. This would greatly increase the trust that all parties can have that the global privacy policies $\mathbb{C}$ hold true.

We also note that, if there are large numbers of agents in our system, we will need a large number of types. In our motivating example, if we have many children and many parents, then we will need types $C_{CHILD_1}(CONSENT)$, $C_{CHILD_2}(CONSENT)$, etc. and a way to ensure that $\pi_{\alpha A}$ accepts terms of type $C_{\alpha'}(A)$ only if $\alpha = \alpha'$, requiring the use of dependent types. For now, this is left as work for the future.

3 Architectures

We now describe the language we use for specifying architectures. This system was inspired by work by le Métayer et al [3] and Barth et al [4].

An *architecture* consists of *agents* who pass *messages* to each other. Each message is a term that can be typed in a *type system*.

Definition 1 (Type System). *A* type system *is given by the following:*

- *A set of* atomic types. *The set of* types *is the defined inductively by:*
  - *Every atomic type is a type.*
  - *If A and B are types, then $A \to B$ is a type.*
- *A set of* constructors, *each with an associated type.*

The set of terms *of each type is then defined inductively by:*

- *Every constructor of type A is a term of type A.*

– *If s is a term of type $A \to B$ and t is a term of type A, then st is a term of type B.*

We write $t : A$ to denote that t is a term of type A.

In the example in Figure 1, the atomic types are $INFO$, $CONSENT$ and $POLICY$. The constructors are $info$ which has type $INFO$, *policy* which has type $POLICY$, and *consent* which has type $CONSENT$. In the example in Figure 2, the architecture has been extended with new atomic types such as $C_{Website}(INFO)$, and new constructors such as p_{POLICY}, which has type $POLICY \to P_{Parent}(POLICY)$.

Definition 2 (Architecture). *Given a type system T, an* architecture $\mathcal{A}$ *over T consists of:*

– *a set* `Ag` *of* agents *or* components*;*
– *for every agent α, a set H_α of constructors that α* initially possesses *or* initially has*;*
– *for every ordered pair of distinct agents (α, β), a set $M_{\alpha\beta}$ of* atomic *types that α may send in a* message *to β.*

We shall write $\alpha \overset{A}{\to} \beta$ to denote that $A \in M_{\alpha\beta}$.

(Note that only terms of atomic type can be passed between agents.)

In the example in section 2, we have $\mathtt{Ag} = \{Child, Website, Parent\}$. The agent *Child* initially possesses the constructor $info$, and *Website* initially possesses *policy*, and *Parent* initially possesses *consent*. We have $M_{Child,Website} = \{INFO\}$; thus, *Child* may send messages of type $INFO$ to *Website*. We also have $M_{Website,Parent} = \{POLICY\}$ and $M_{Parent,Website} = \{CONSENT\}$.

We will use lower-case Greek letters α, β, … for agents, lower-case Roman letters s, t, … for terms, and capital Roman letters A, B, … for types. The letter c is reserved for constructors.

Let us say that an agent α can *compute* terms of type A iff it possesses a constructor of type $B_1 \to \cdots \to B_n \to A$ for some $B_1, \ldots, B_n$.

Definition 3. *Let $\mathcal{A}$ be an architecture.*

1. *An* event *or* message *is an expression of the form $\alpha \overset{t:A}{\to} \beta$, to be read as '$\alpha$ passes the term t of type A to β.'*
2. *A* trace *τ is a finite sequence of events.*
3. *A* judgement *is an expression of the form $\tau \vdash \alpha \ni t : A$, which we read as "After the trace τ, α has the term t of type A."*

We write τ_1, τ_2 for the concatenation of traces τ_1 and τ_2. We write $\tau_1 \sqsubseteq \tau'$ iff τ_1 is a prefix of τ', i.e. there exists τ_2 such that $\tau' = \tau_1, \tau_2$.

The *derivable* judgements are given by the rules of deduction in Figure 3. We say that τ is a *valid* trace through $\mathcal{A}$ iff $\tau \vdash \alpha \ni t : A$ is derivable for some α, t, A. We say that an agent α *possesses* a term of type A after τ, and write

$$(init)\,\frac{}{\vdash \alpha \ni c : A}\;(c : A \in H_\alpha) \qquad (message_1)\,\frac{\tau \vdash \alpha \ni t : A}{\tau, \alpha \overset{t:A}{\to} \beta \vdash \beta \ni t : A}\;(A \in M_{\alpha\beta})$$

$$(message_2)\,\frac{\tau \vdash \alpha \ni t : A \qquad \tau \vdash \gamma \ni s : C}{\tau, \alpha \overset{t:A}{\to} \beta \vdash \gamma \ni s : C}\;(A \in M_{\alpha\beta})$$

$$(func)\,\frac{\tau \vdash \alpha \ni f : A \to B \qquad \tau \vdash \alpha \ni t : A}{\tau \vdash \alpha \ni ft : B}$$

Fig. 3. Rules of Deduction

$\tau \vdash \alpha \ni A$, iff there exists a term t such that $\tau \vdash \alpha \ni t : A$. We say that there *exists* a term of type A after τ, and write $\tau \vdash A$, iff $\tau \vdash \alpha \ni t : A$ for some α, t.

The rule (init) states that, if α initially possesses c, then α possesses c in the initial state. The rule (func) states that, if an agent possesses both a function f and term t of the appropriate types, it may compute the term ft. The rule (message$_1$) states that, after α has sent t to β, then β possesses t. The rule (message$_2$) states that, if γ possesses s before α sends a message to β, then γ still possesses s after the message is sent.

3.1 Metatheorems

We can establish the basic properties that our typing system satisfies.

Lemma 1.

1. ***Weakening*** *Suppose $\tau_1 \vdash \alpha \ni t : A$ and τ_1, τ_2 is a valid trace. Then $\tau_1, \tau_2 \vdash \alpha \ni t : A$.*
2. *If $\tau_1, \alpha \overset{t:A}{\to} \beta, \tau_2$ is a valid trace, then $A \in M_{\alpha\beta}$, and $\tau_1 \vdash \alpha \ni t : A$.*
3. ***Generation*** *Suppose $\tau \vdash \beta \ni t : B$. Then there exist terms $t_1 : A_1, \ldots, t_m : A_m$ $(m \geq 0)$ and agents $\alpha_1, \ldots, \alpha_n$ $(n \geq 1)$ such that $t \equiv ft_1 \cdots t_m$, $\beta = \alpha_n$, and the following events occur in τ in order:*

$$\alpha_1 \ni f : A_1 \to \cdots \to A_n \to B,\ \alpha_1 \overset{t:B}{\to} \alpha_2,\ \cdots,\ \alpha_{n-1} \overset{t:B}{\to} \alpha_n$$

   Further, we have $\tau \vdash \alpha_1 \ni t_1 : A_1, \ldots, \tau \vdash \alpha_1 \ni t_m : A_m$.
4. *If $\tau \vdash \beta \ni t : B$, then either β can compute B, or there is an event $\alpha \overset{t:B}{\to} \beta$ in τ for some α.*

Intuitively, Generation says that if agent β possesses a piece of data of type B, then it must have been computed by an agent α_1 that can compute terms of type B, and then passed to β in a sequence of messages.

The proofs of the first three properties are by straightforward induction on derivations. Part 4 follows easily from part 3.

4 The First Algorithm

In the rest of this paper, we will consider different sets of *constraints* that we may wish to place on our architectures. In each case, we shall show how, given an architecture $\mathcal{A}$ and a set of constraints $\mathbb{C}$, we can construct an architecture $\mathcal{B}$, which we call a *safe* architecture, that extends $\mathcal{A}$ and satisfies all the constraints.

For our first algorithm, we consider the following constraints:

Definition 4 (Constraint).

1. *A* negative constraint *has the form $\alpha \ni A \Rightarrow B$, where A and B are atomic types. We read it as: "If α receives a message of type A, then a term of type B must have previously been created." A trace τ* complies *with this constraint iff, for every $\tau_1 \sqsubseteq \tau$, if $\tau_1 \vdash \alpha \ni t : A$ for some t, then $\tau_1 \vdash \beta \ni s : B$ for some β, s.*
2. *A* positive constraint *has the form $\mathtt{Pos}(\alpha, A)$, where A is an atomic type. We read it as: "It must be possible for α to have a term of type A." A trace τ* complies *with this constraint iff $\tau \vdash \alpha \ni t : A$ for some term t.*

Note To understand part 1 of this definition, note that, if it is possible to create a term $t : A$ without first creating a term $s : B$, then there is a trace τ such that $\tau \vdash \alpha \ni t : A$ for some α, and $\tau \nvdash \beta \ni s : B$ for all β. Thus, the condition "For every $\tau_1 \sqsubseteq \tau$, if $\tau_1 \vdash \alpha \ni t : A$ for some t, then $\tau_1 \vdash \beta \ni s : B$ for some β, s" captures the idea "If α receives a message of type A, then a term of type B must have previously been created."

Example Consider an accountancy firm collecting personal data from the employees of a company in order to prepare a tax report. The principle of *data minimization* [13, Section 25] states that the accountancy firm should collect only the data that is necessary for this purpose. We can model this as follows: assume there are two types of tax return that can be prepared, TR_A and TR_B. Let *Employee* initially possess $a : A$ and $b : B$, where a is required to prepare TR_A and b is required to prepare TR_B. The company can send requests Q_A and Q_B to *Accountancy*, requesting a tax return of one of the two types. We could then write constraints *Accountancy* $\ni A \Rightarrow TR_A$ and *Accountancy* $\ni B \Rightarrow TR_B$ to express that the accountancy firm may only possess an employee's personal data if it is necessary for a tax return that it has been requested to prepare.

We now construct the type system that the safe architecture will use:

Definition 5 (Safe Type System). *Let $\mathcal{T}$ be a type system and* $\mathtt{Ag}$ *a set of agents. Let $\mathbb{C}$ be a finite set of negative constraints over $\mathcal{T}$ and* $\mathtt{Ag}$*. The* safe *type system $\mathcal{T}_{\mathbb{C}}$ is defined as follows.*

- *The atomic types of $\mathcal{T}_{\mathbb{C}}$ are the atomic types of $\mathcal{T}$ together with, for every agent $\alpha \in \mathtt{Ag}$ and atomic type A in $\mathcal{T}$, a type $C_\alpha(A)$, the type of* certified *terms of type A that may only be read by α.*
- *Every constructor of $\mathcal{T}$ is a constructor of $\mathcal{T}_{\mathbb{C}}$.*

– *For every $\alpha \in \mathtt{Ag}$ and type A of $\mathcal{T}$, let the constraints in $\mathbb{C}$ that begin with '$\alpha \ni A$' be*

$$\alpha \ni A \Rightarrow B_1, \ldots, \alpha \ni A \Rightarrow B_n \ .$$

Then the following are constructors of $\mathcal{T}_{\mathbb{C}}$:

$m_{\alpha A}^{\beta_1 \cdots \beta_n} : A \to C_{\beta_1}(B_1) \to \cdots \to C_{\beta_n}(B_n) \to C_\alpha(A)$ *for all* $\beta_1, \ldots, \beta_n \in \mathtt{Ag}$;
$\pi_{\alpha A} : C_\alpha(A) \to A$

The intention is that $m_{\alpha A}^{\beta_1 \cdots \beta_n}$ constructs a term of type $C_\alpha(A)$ out of a term of type A and n other terms which prove that the preconditions to $\alpha \ni A$ are all satisfied. The constructor $\pi_{\alpha A}$ then extracts the term of type A again.

Using the type system, we can state a set of conditions that guarantee that an architecture satisfies the negative constraints in $\mathbb{C}$.

Theorem 1. *Let $\mathcal{T}$ be a type system, $\mathtt{Ag}$ a set of agents, and $\mathbb{C}$ a set of negative constraints over $\mathcal{T}$ and $\mathtt{Ag}$. Let $\mathcal{B}$ be an architecture over $\mathcal{T}_{\mathbb{C}}$ with set of agents $\mathtt{Ag}'$, where $\mathtt{Ag} \subseteq \mathtt{Ag}'$. Suppose there is a partition $\{\mathcal{P}_\alpha \subseteq \mathtt{Ag}'\}_{\alpha \in \mathtt{Ag}}$ of $\mathtt{Ag}'$ indexed by $\mathtt{Ag}$ such that:*

1. *$\alpha \in \mathcal{P}_\alpha$ for all $\alpha \in \mathtt{Ag}$;*
2. *If $\beta \xrightarrow{A} \beta'$ and β, β' are in different sets of the partition, then A has the form $C_\gamma(B)$ for some γ, B;*
3. *If β possesses $\pi_{\alpha A}$ then $\beta \in \mathcal{P}_\alpha$;*
4. *For every constraint $\alpha \ni A \Rightarrow B$ in $\mathbb{C}$, if an agent $\beta \in \mathcal{P}_\alpha$ possesses a constructor with target A, then this constructor is $\pi_{\alpha A}$.*

Then every trace through $\mathcal{B}$ satisfies every negative constraint in $\mathbb{C}$.

The intuition behind the premises is this: the partition divides the system into parts. The part $\mathcal{P}_\alpha$ is the only part of the system that is allowed to look inside a term of type $C_\alpha(A)$ and extract the underlying term of type A. Only certified terms may be passed between the parts. Thus, the only way for an agent in $\mathcal{P}_\alpha$ to possess a term of type A is either for it to be computed within $\mathcal{P}_\alpha$, or for a term of type $C_\alpha(A)$ to be passed in from another part of the system.

Proof. Let τ be any trace through $\mathcal{B}$ and let $\alpha \ni A \Rightarrow B$ be one of the constraints in $\mathbb{C}$. We must show that, if $\tau \vdash \alpha \ni t : A$, then $\tau \vdash B$. We shall prove the more general result:

If $\tau \vdash \beta \ni t : A$ for some $\beta \in \mathcal{P}_\alpha$, then $\tau \vdash B$.

So suppose $\tau \vdash \beta \ni t : A$ for some $\beta \in \mathcal{P}_\alpha$. We may also assume without loss of generality that τ is the shortest trace for which this is true. By Generation and the minimality of τ, β possesses a constructor with target A. By our hypotheses, this is $\pi_{\alpha A}$, and $t = \pi_{\alpha A}(t')$ for some t'. Hence $\tau \vdash \beta \ni t' : C_\alpha(A)$ for some t'.

Now, looking at the construction of $\mathcal{T}_{\mathbb{C}}$, the only constructor with target $C_\alpha(A)$ is

$$m_{\alpha A}^{\beta_1 \cdots \beta_n} : A \to C_{\beta_1}(B_1) \to \cdots \to C_{\beta_n}(B_n) \to C_\alpha(A) \ .$$

So applying Generation again, we must have $t \equiv m_{\alpha A}^{\beta_1 \cdots \beta_n} s t_1 \cdots t_n$ and there must be an agent γ which possesses $m_{\alpha A}^{\beta_1 \cdots \beta_n}$ with

$$\tau \vdash \gamma \ni s : A, \quad \tau \vdash \gamma \ni t_1 : B_1, \ldots, \tau \vdash \gamma \ni t_n : B_n \ .$$

Now, B is one of the types $B_1, \ldots, B_n$; let it be B_i. Then $\tau \vdash \gamma \ni C_{\beta_i}(B)$. By similar reasoning, there must be an agent δ that possesses one of the constructors $m_{\beta_i B}$, and $\tau \vdash \delta \ni B$. □

We are now ready to construct the safe architecture.

Algorithm 1 *Given an architecture $\mathcal{A}$ and a finite set of constraints $\mathbb{C}$, construct the architecture* $\mathtt{Safe}(\mathcal{A}, \mathbb{C})$ *as follows:*

1. *The agents of* $\mathtt{Safe}(\mathcal{A}, \mathbb{C})$ *are the agents of $\mathcal{A}$ together with, for every agent α of $\mathcal{A}$, an agent I_α, which we call the* interface *to α.*
2. *The type system of* $\mathtt{Safe}(\mathcal{A}, \mathbb{C})$ *is $\mathcal{T}_{\mathbb{C}}$.*
3. *If an agent α possesses a constructor c in $\mathcal{A}$, then α possesses c in* $\mathtt{Safe}(\mathcal{A}, \mathbb{C})$.
4. *For every type A of $\mathcal{A}$, let the negative constraints that begin with $\alpha \ni A$ be*

$$\alpha \ni A \Rightarrow B_1, \ldots, \alpha \ni A \Rightarrow B_n \ .$$

   – Every *interface I_γ possesses $m_{\alpha A}^{\beta_1 \cdots \beta_n}$ for all $\beta_1, \ldots, \beta_n$.*
   – *I_α posseses $\pi_{\alpha A}$*
5. *For every atomic type A, the agents α and I_α may send A to each other.*
6. *Any two interfaces may send messages of type $C_\alpha(A)$ to each other for any α, A.*

Thus, in order to construct a certified term of type A readable by α, an interface must first obtain certified terms of all the types which the constraints require. The only way α can receive a term of type A is through its interface obtaining a term of type $C_\alpha(A)$. Interfaces may pass certified terms between each other at will. An agent and its interface may exchange uncertified terms at will.

Theorem 2. *Let $\mathcal{A}$ be an architecture and $\mathbb{C}$ a set of constraints. Suppose that:*

1. *For every negative constraint $\alpha \ni A \Rightarrow \beta \ni B$ in $\mathbb{C}$, we have that α cannot compute terms of type A.*
2. *For every positive constraint* $\mathtt{Pos}(\alpha, A) \in \mathbb{C}$, *there exists a trace through $\mathcal{A}$ that satisfies* $\mathtt{Pos}(\alpha, A)$ *and all the negative constraints in $\mathbb{C}$.*

Then the architecture $\mathtt{Safe}(\mathcal{A}, \mathbb{C})$ *has the following properties:*

1. *Every trace through* $\mathtt{Safe}(\mathcal{A}, \mathbb{C})$ *satisfies every negative constraint in $\mathbb{C}$.*
2. *For every positive constraint* $\mathtt{Pos}(\alpha, A) \in \mathbb{C}$, *there exists a trace through* $\mathtt{Safe}(\mathcal{A}, \mathbb{C})$ *that satisfies* $\mathtt{Pos}(\alpha, A)$.

Proof. Part 1 follows from the previous theorem, taking $\mathcal{P}_\alpha = \{\alpha, I_\alpha\}$.

We now show that $\mathtt{Safe}(\mathcal{A}, \mathbb{C})$ has the following property. Part 2 of the theorem follows immediately.

If $\tau \vdash \alpha \ni t : A$ in $\mathcal{A}$, A is an atomic type, and τ satisfies every negative constraint in $\mathbb{C}$, then there exists a valid trace τ' through $\texttt{Safe}(\mathcal{A}, \mathbb{C})$ such that $\tau' \vdash \alpha \ni t : A$ and $\tau' \vdash I_\alpha \ni t' : C_\alpha(A)$ for some t'.

The proof is by induction on τ, then on the derivation of $\tau \vdash \alpha \ni t : A$. We deal here with the case where the last rule in the derivation was ($message_1$):

$$\frac{\tau \vdash \beta \ni t : A}{\tau, \beta \overset{t:A}{\to} \alpha \vdash \alpha \ni t : A} \ .$$

By the induction hypothesis, there exists τ' such that $\tau' \vdash_{\texttt{Safe}(\mathcal{A},\mathbb{C})} \beta \ni t : A$. By the construction of $\texttt{Safe}(\mathcal{A}, \mathbb{C})$, we have $A \in C_{\beta I_\beta}$ and $A \in C_{I_\alpha \alpha}$. Hence $\tau, \beta \overset{t:A}{\to} I_\beta \vdash_{\texttt{Safe}(\mathcal{A},\mathbb{C})} I_\beta \ni t : A$.

Now, let the negative constraints in $\mathbb{C}$ that begin with $\alpha \ni A$ be $\alpha \ni A \Rightarrow B_1, \ldots, \alpha \ni A \Rightarrow B_n$. By hypothesis, $\tau, \beta \overset{t:A}{\to} \alpha$ satisfies all these constraints. Therefore, $\tau \vdash_{\mathcal{A}} B_1, \ldots, \tau \vdash_{\mathcal{A}} B_n$.

Hence, by the induction hypothesis, there exists τ'' such that $\tau'' \vdash_{\texttt{Safe}(\mathcal{A},\mathbb{C})} B_1, \ldots, \tau'' \vdash_{\texttt{Safe}(\mathcal{A},\mathbb{C})} B_n$. Therefore,

$$\tau'' \vdash_{\texttt{Safe}(\mathcal{A},\mathbb{C})} I_{\beta_1} \ni t_1 : C_{\beta_1}(B_1), \ldots, \tau'' \vdash_{\texttt{Safe}(\mathcal{A},\mathbb{C})} I_{\beta_n} \ni t_n : C_{\beta_n}(B_n) \ ,$$

for some $t_1, \ldots, t_n$. By Weakening, we may assume $\tau' \sqsubseteq \tau''$.

After extending τ'' by passing $t_1, \ldots, t_n$ as messages to I_β, we have that I_β can construct a term of type $C_\alpha(A)$. After passing this term to I_α, we have that $I_\alpha(A)$ possesses a term of type $C_\alpha(A)$. From this, it can construct a term of type A which it may then pass to α, completing the required trace. □

5 The Second Algorithm

Supposing it is important to us, not merely that a piece of data has been created, but that a particular agent has seen it. We can extend our system to handle this type of constraint as follows.

Definition 6. *In this section of the paper:*

- *a* negative constraint *is an expression of the form $\alpha \ni A \Rightarrow \beta \ni B$. A trace τ satifies this constraint iff, for every $\tau' \sqsubseteq \tau$, if $\tau' \vdash \alpha \ni A$ then $\tau_1 \vdash \beta \ni B$.*
- *Positive constraints are as in Section 4.*

Note If $(\alpha, A) \neq (\beta, B)$, then the constraint $\alpha \ni A \Rightarrow \beta \ni B$ is to be read as "if α possesses a term of type A, then β must previously have possessed a term of type B". (The condition $\alpha \ni A \Rightarrow \alpha \ni A$ is trivial.)

We show how to extend a given architecture $\mathcal{A}$ to an architecture that uses the new privacy-safe type system. Unfortunately, we have not found a way to do this that requires no modifications to the agents in $\mathcal{A}$. We present below (Algorithm 2) an algorithm that requires modifications which we expect would be minor in practice, and discuss in Section 5.2 ways in which this situation could be improved in future work.

Definition 7. *Given a type system $\mathcal{T}$, a set of agents* $\mathtt{Ag}$*, and a set of negative constraints $\mathbb{C}$ over $\mathcal{T}$ and* $\mathtt{Ag}$*, define the type system $\mathcal{T}_{\mathbb{C}}$ as follows.*

- *The types of $\mathcal{T}_{\mathbb{C}}$ are the types of $\mathcal{T}$ together with, for every agent α and atomic type A of $\mathcal{T}$, a type $C_\alpha(A)$ and a type $P_\alpha(A)$. (Intuition: a term $C_\alpha(A)$ is a* certified *term of type A that α is permitted to read. A term $P_\alpha(A)$ is* proof *that α has held a term of type A.)*
- *Every constructor of $\mathcal{T}$ is a constructor of $\mathcal{T}_{\mathbb{C}}$.*
- *For every agent α and type A, let the negative contraints in $\mathbb{C}$ that begin with $\alpha \ni A$ be*
$$\alpha \ni A \Rightarrow \beta_1 \ni B_1, \ldots, \alpha \ni A \Rightarrow \beta_n \ni B_n \enspace .$$
Then the following are constructors of $\mathcal{T}_{\mathbb{C}}$:
$$\begin{aligned} m_{\alpha A} &: A \to P_{\beta_1}(B_1) \to \cdots \to P_{\beta_n}(B) \to C_\alpha(A) \\ \pi_{\alpha A} &: C_\alpha(A) \to A \\ p_{\alpha A} &: A \to P_\alpha(A) \end{aligned}$$

Theorem 3. *Let $\mathcal{T}$ be a type system,* $\mathtt{Ag}$ *a set of agents, and $\mathbb{C}$ a set of negative constraints over $\mathcal{T}$ and* $\mathtt{Ag}$*. Let $\mathcal{B}$ be an architecture over $\mathcal{T}_{\mathbb{C}}$ with set of agents $\mathtt{Ag}'$, where $\mathtt{Ag} \subseteq \mathtt{Ag}'$. Suppose that there is a partition $\{\mathcal{P}_\alpha\}_{\alpha \in \mathtt{Ag}}$ of the agents of $\mathcal{B}$ such that:*

- *$\alpha \in \mathcal{P}_\alpha$;*
- *If $\beta \overset{A}{\to} \beta'$ and β and β' are in different sets in the partition, then A has either the form $C_\gamma(T)$ or $P_\gamma(T)$;*
- *If β initially possesses $\pi_{\alpha A}$ then $\beta \in \mathcal{P}_\alpha$;*
- *If β initially possesses $p_{\alpha A}$ then β cannot compute A.*
- *If β initially possesses $p_{\alpha A}$ and $\gamma \overset{A}{\to} \beta$ then $\gamma = \alpha$.*

Then every trace through $\mathcal{B}$ satisfies every constraint in $\mathbb{C}$.

Proof. Let τ be a trace through $\mathcal{B}$ and $\alpha \ni A \Rightarrow \beta \ni B$ be a constraint in $\mathbb{C}$. We must show that, if $\tau \vdash \alpha \ni A$, then $\tau \vdash \beta \ni B$. We shall prove the more general result:

If $\tau \vdash \gamma \ni A$ for any $\gamma \in \mathcal{P}_\alpha$, then $\tau \vdash \beta \ni B$.

So suppose $\tau \vdash \gamma \ni A$ for some $\gamma \in \mathcal{P}_\alpha$. We may assume without loss of generality that τ is the shortest such trace. By Generation and the minimality of τ, γ must possess a constructor with target A. By our hypotheses, this is $\pi_{\alpha A}$. Hence $\tau \vdash \gamma \ni t : C_\alpha(A)$ for some t. Now, let the constraints in $\mathbb{C}$ that begin with $\alpha \ni A$ be

$$\alpha \ni A \Rightarrow \beta_1 \ni B_1, \quad \cdots, \quad \alpha \ni A \Rightarrow \beta_n \ni B_n \enspace .$$

Applying Generation, we must have $t \equiv m_{\alpha A} s t_1 \cdots t_n$, and there must be an agent γ' that possesses $m_{\alpha A}$ such that

$$\tau \vdash \gamma' \ni s : A, \quad \tau \vdash \gamma' \ni t_1 : P_{\beta_1}(B_1), \ldots, \tau \vdash \gamma' \ni t_n : P_{\beta_n}(B_n) \enspace .$$

Now, there is some i such that $\beta_i = \beta$ and $B_i = B$. We have $\tau \vdash \gamma' \ni t_i : P_\beta(B)$. Since a term of type $P_\beta(B)$ has been constructed, it must be that $\tau \vdash \beta \ni B$, as required.

We now show again how, given an architecture $\mathcal{A}$, we can construct an architecture that is privacy-safe.

Algorithm 2 *Given an architecture $\mathcal{A}$ and a finite set of constraints $\mathbb{C}$, construct the architecture* $\mathtt{Safe}(\mathcal{A}, \mathbb{C})$ *as follows:*

1. *The agents of* $\mathtt{Safe}(\mathcal{A}, \mathbb{C})$ *are the agents of $\mathcal{A}$ together with, for every agent α of $\mathcal{A}$:*
   - *an agent I_α, which we call the* input interface *to α;*
   - *an agent O_α, which we call the* output interface *to α*
2. *The type system of* $\mathtt{Safe}(\mathcal{A}, \mathbb{C})$ *is $\mathcal{T}_\mathbb{C}$.*
3. *If an agent α has a constructor c in $\mathcal{T}$, then it has the constructor c in $\mathcal{T}_\mathbb{C}$.*
4. *For any agent α and type A:*
   - *Every output interface O_γ possesses $m_{\alpha A}$*
   - *I_α possesses $\pi_{\alpha A} : C_{\alpha A} \to A$*
   - *O_α possesses $p_{\alpha A} : A \to P_{\alpha A}$*
5. *For any atomic type A of $\mathcal{T}$, I_α may send A to α, and α may send A to O_α.*
6. *Any two interfaces may send messages of type $C_\alpha(A)$ or $P_\alpha(A)$ to each other for any α, A.*

Theorem 4. *Let $\mathcal{A}$ be an architecture and $\mathbb{C}$ a set of constraints. Suppose that:*

1. *For every negative constraint $\alpha \ni A \Rightarrow B$ in $\mathbb{C}$, we have that α cannot compute terms of type A.*
2. *For every positive constraint* $\mathtt{Pos}(\alpha, A) \in \mathbb{C}$, *there exists a trace through $\mathcal{A}$ that satisfies* $\mathtt{Pos}(\alpha, A)$ *and all the negative constraints in $\mathbb{C}$.*

Then the architecture $\mathtt{Safe}(\mathcal{A}, \mathbb{C})$ *has the following properties:*

1. *Every trace through* $\mathtt{Safe}(\mathcal{A}, \mathbb{C})$ *satisfies every negative constraint in $\mathbb{C}$.*
2. *For every positive constraint* $\mathtt{Pos}(\alpha, A) \in \mathbb{C}$, *there exists a trace through* $\mathtt{Safe}(\mathcal{A}, \mathbb{C})$ *that satisfies* $\mathtt{Pos}(\alpha, A)$.

Proof. Part 1 follows from Theorem 3, taking $P_\alpha = \{\alpha, I_\alpha, O_\alpha\}$.

We shall now prove the following property, from which part 2 of the theorem follows.

> If $\tau \vdash \alpha \ni t : A$ in $\mathcal{A}$ and τ satisfies every negative constraint in $\mathbb{C}$, then there exists a trace τ' through $\mathtt{Safe}(\mathcal{A}, \mathbb{C})$ such that $\tau' \vdash \alpha \ni t : A$.

The proof is by induction on τ, then on the derivation of $\tau \vdash \alpha \ni t : A$. We deal here with the case where the final step in the derivation is an instance of ($message_1$):

$$\frac{\tau \vdash \beta \ni t : A}{\tau, \beta \overset{t:A}{\to} \alpha \vdash \alpha \ni t : A}$$

By the induction hypothesis, there is a trace τ' such that $\tau' \vdash_{\mathtt{Safe}(\mathcal{A},\mathbb{C})} \alpha \ni A$. Let the negative constraints beginning with $\alpha \ni A$ be

$$\alpha \ni A \Rightarrow \beta_1 \ni B_1, \quad \ldots, \quad \alpha \ni A \Rightarrow \beta_n \ni B_n \ .$$

Then, by hypothesis,

$$\tau, \beta \overset{t:A}{\to} \alpha \vdash_{\mathcal{A}} \beta_1 \ni B_1, \quad \cdots, \quad \tau, \beta \overset{t:A}{\to} \alpha \vdash_{\mathcal{A}} \beta_n \ni B_n \ .$$

Using the fact that $(\alpha, A) \neq (\beta_i, B_i)$ for all i, the last step in each of these derivations must have been $(\mathit{message}_2)$. Therefore,

$$\tau \vdash_{\mathcal{A}} \beta_1 \ni B_1, \quad \cdots, \quad \tau \vdash_{\mathcal{A}} \beta_n \ni B_n \ .$$

We may therefore apply the induction hypothesis to obtain traces $\tau_1, \ldots, \tau_n$ such that

$$\tau_1 \vdash_{\mathtt{Safe}(\mathcal{A},\mathbb{C})} \beta_1 \ni t_1 : B_1, \quad \cdots, \quad \tau_n \vdash_{\mathtt{Safe}(\mathcal{A},\mathbb{C})} \beta_n \ni t_n : B_n \ .$$

Now, let τ'' be the trace $\tau', \tau_1, \ldots, \tau_n$ followed by these events:

$$\begin{aligned}
&\beta \xrightarrow{t:A} O_\beta, \beta_1 \xrightarrow{t_1:B_1} O_{\beta_1}, \cdots, \beta_n \xrightarrow{t_n:B_n} O_{\beta_n}, \\
&O_{\beta_1} \xrightarrow{p_{\beta_1 B_1} t_1} O_\beta, \cdots, O_{\beta_n} \xrightarrow{p_{\beta_n, B_n} t_n} O_\beta, \\
&O_\beta \xrightarrow{c_{\alpha A} t (p_{\beta_1 B_1} t_1) \cdots (p_{\beta_n B_n} t_n)} I_\alpha, \\
&I_\alpha \xrightarrow{\pi_{\beta A}(c_{\alpha A} t (p_{\beta_1 B_1} t_1) \cdots (p_{\beta_n B_n} t_n))} \alpha
\end{aligned}$$

(Informally: the agent O_β collects the term of type A from β and all the necessary proofs, assembles the term of type $C_{\alpha A}$, and passes it to I_α, who decodes it with $\pi_{\alpha A}$ and passes the value of A to α.)

We thus have $\tau'' \vdash \alpha \ni A$ in $\mathtt{Safe}(\mathcal{A}, \mathbb{C})$, as required. □

5.1 Example Revisited

We return to the example we presented in Section 2. We are now ready to formulate our third, positive constraint. We want to ensure it is possible for the website to receive the child's information once all legal requirements have been met. So the privacy constraints that we require for this architecture are:

Negative Constraint $\mathit{Website} \ni \mathit{INFO} \Rightarrow \mathit{Website} \ni \mathit{CONSENT}$
Negative Constraint $\mathit{Website} \ni \mathit{CONSENT} \Rightarrow \mathit{Parent} \ni \mathit{POLICY}$
Positive Constraint $\mathtt{Pos}(\mathit{Website}, \mathit{INFO})$

We can verify that the first constraint holds. The child can send the protected info to the interface OChild, but it cannot then be sent to another agent unless OChild receives a term of type $P(\mathit{CONSENT})$. And for a term of type

$P(CONSENT)$ to be constructed, the parent must have sent consent to the website (via $OParent$ and $IWebsite$).

We can also verify that, in the architecture in Figure 2, it is possible for the website to send the privacy policy to the parent, the parent to send consent to the website, and the child to send the protected info to the website. Formally, we describe a valid trace τ through the architecture that represents this sequence of events. The trace τ begins

$$\begin{array}{rcl} Website & \xrightarrow{policy:POLICY} & OWebsite, \\ OWebsite & \xrightarrow{mPOLICY(policy):C(POLICY)} & IParent, \\ IParent & \xrightarrow{\pi POLICY(mPOLICY(policy)):POLICY} & Parent, \\ Parent & \xrightarrow{consent:CONSENT} & OParent, \\ Parent & \xrightarrow{\pi POLICY(mPOLICY(policy)):POLICY} & OParent \end{array}$$

Let $p = \pi POLICY(mPOLICY(policy)$. The trace τ continues:

$$\begin{array}{rcl} OParent & \xrightarrow{mCONSENT(consent,p):C(CONSENT)} & IWebsite, \\ IWebsite & \xrightarrow{\pi CONSENT(mCONSENT(consent,p)):CONSENT} & Website, \\ Website & \xrightarrow{\pi CONSENT(mCONSENT(consent,p)):CONSENT} & OWebsite, \end{array}$$

Let $c = \pi CONSENT(mCONSENT(consent, p))$. The trace τ continues:

$$\begin{array}{rcl} OWebsite & \xrightarrow{pCONSENT(c):P(CONSENT)} & OChild, \\ Child & \xrightarrow{info:INFO} & OChild, \\ OChild & \xrightarrow{mINFO(info,pCONSENT(c)):C(INFO)} & IWebsite, \\ IWebsite & \xrightarrow{\pi INFO(mINFO(info,pCONSENT(c))):INFO} & Website \end{array}$$

This ends the trace τ which verifies that it is possible for $Website$ to receive a term of type $INFO$.

5.2 Note

In Fig. 2, we have had to modify the agents from Fig. 1. The agent $Parent$ needs to be able to output messages of type $POLICY$, and $Website$ needs to be able to output messages of type $CONSENT$. We believe these would be minor changes in practice. However, this is still unfortunate, because as discussed in the Introduction, we want our algorithms to apply in cases in which we are unable to change the source code of the agents in $\mathcal{A}$.

In practice, we could implement this by allowing *IParent* to send $POLICY$ to *OParent*, and *IWebsite* to send $POLICY$ to *OWebsite*, and adding the following local constraints to their behaviour:

– If *IParent* sends $t : POLICY$ to *OParent*, then *IParent* must previously have sent t to *Parent*.
– If *IWebsite* sends $t : POLICY$ to *OWebsite*, then *IWebsite* must previously have sent t to *Website*.

Obtaining a formal proof of correctness for this construction requires an architecture language in which this sort of local constraint can be expressed, and we leave this for future work.

6 Related Work

Le Métayer et al. [3, 2, 7, 6] have described several languages for describing architectures and deciding privacy-related properties over them. Barth et al. [4] also give a formal definition of architectures, and show how to decide properties defined in temporal logic. Our work was heavily inspired by these systems; however, our aim was to give a method to design an architecture starting from a set of privacy properties, and not to decide whether a property holds of a given architecture.

Basin et al. [5] show how to describe privacy policies in metric first-order temporal logic (MFOTL), and how to build a component that monitors in real-time whether these policies are being violated. Nissenbaum et al [4] also describe privacy policies using linear temporal logic (LTL), and this has inspired a lot of research into systems such as P-RBAC, which enforces low-level privacy-related conditions at run-time [17]. Most of this research has concentrated on verifying at run-time whether or not a given action is permitted by a given set of privacy policies. The work presented here concentrates instead on design-time, and ensures that a high-level privacy policy is followed, no matter what actions each individual component performs with the data it receives, as long as all messages follow the given type system.

Jeeves [20] is a constraint functional language motivated by separating business logic and confidentially concerns. We could implement our (architectural) constraints in Jeeves, but would no longer have static guarantees. Other work in formal methods for privacy includes static analysis of source code [11, 15] and refinement techniques for deriving low-level designs from high-level designs in a way that preserves privacy properties [1, 10]. These approaches complement ours well, addressing properties for individual components that cannot be expressed in our constraint language, while our algorithms provide formal guarantees of global properties of the system as a whole.

Other work in formal methods for privacy has tended to concentrate either on static analysis of source code [15, 11] or on refinement techniques for deriving low-level designs from high-level designs in a way that preserves privacy properties [1, 10]. These approaches should complement ours well, providing formal guarantees for individual components of properties that cannot be expressed in our constraint language, while our algorithms provide formal guarantees of global properties of the system as a whole.

7 Conclusion

We have given two algorithms which take an architecture, and a set of constraints on that architecture, and show how the architecture may be extended in such a way that we can produce a formal proof that the negative constraints hold on every trace through the architecture, and the positive constraints are satisfiable. Moreover, we do not need to read or modify the source code of the components from the original architecture in order to do this. We believe this is a promising approach to designing large, complex systems, with many different parts designed and maintained by different people, such that we can provide a formal proof of privacy-relevant properties.

For the future, we wish to expand the language that may be used for our constraints, for example by allowing the designer to express constraints using propositional, predicate or temporal logic. We hope then to express other properties that are desirable for privacy, such as the obligation to delete data. This will require in turn expanding our type systems $\mathcal{T}_{\mathbb{C}}$. We also plan to construct a prototype implementation of the interfaces described in this paper.

References

1. Alur, R., Černỳ, P., Zdancewic, S.: Preserving secrecy under refinement. Automata, Languages and Programming pp. 107–118 (2006)
2. Antignac, T., le Métayer, D.: Privacy architectures: Reasoning about data minimisation and integrity. In: Security and Trust Management — 10th International Workshop, STM 2014. pp. 17–32 (2014)
3. Antignac, T., le Métayer, D.: Privacy by design: From technologies to architectures. In: Privacy Technologies and Policy — Second Annual Privacy Forum, APF 2014. pp. 1–17 (2014)
4. Barth, A., Datta, A., Mitchell, J.C., Nissenbaum, H.: Privacy and contextual integrity: Framework and applications. In: IEEE Symposium on Security and Privacy. pp. 184–198 (2006)
5. Basin, D., Klaedtke, F., Müller, S.: Monitoring security policies with metric first-order temporal logic. In: ACM SACMAT '10 (2010)
6. Butin, D., Chicote, M., le Métayer, D.: Log design for accountability. In: IEEE Symposium on Security and Privacy Workshops. pp. 1–7 (2013)
7. Butin, D., Chicote, M., le Métayer, D.: Reloading data protection, pp. 343–369 (2014)
8. Cavoukian, A.: Privacy by design. IEEE Technology and Society Magazine 31(4), 18–19 (2012)

9. Cavoukian, A., Stoddart, J., Dix, A., Nemec, I., Peep, V., Shroff, M.: Resolution on privacy by design. In: 32nd International Conference of Data Protection and Privacy Commissioners (2010)
10. Clarkson, M.R., Schneider, F.B.: Hyperproperties. Journal of Computer Security 18(6), 1157–1210 (2010)
11. Cortesi, A., Ferrara, P., Pistoia, M., Tripp, O.: Datacentric semantics for verification of privacy policy compliance by mobile applications. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 61–79. Springer (2015)
12. Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Transactions on Computer Systems (TOCS) 32(2), 5 (2014)
13. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). Official Journal of the European Union L119, 1–88 (May 2016), http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=OJ:L:2016:119:TOC
14. Federal Trade Commission: Protecting consumer privacy in an era of rapid change. FTC report (2012)
15. Ferrara, P., Tripp, O., Pistoia, M.: Morphdroid: fine-grained privacy verification. In: Proceedings of the 31st Annual Computer Security Applications Conference. pp. 371–380. ACM (2015)
16. Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif: Java information flow (2001)
17. Ni, Q., Bertino, E., Lobo, J., Brodie, C., Karat, C.M., Karat, J., Trombeta, A.: Privacy-aware role-based access control. ACM Transactions on Information and System Security (TISSEC) 13(3), 24 (2010)
18. Pottier, F., Simonet, V.: Information flow inference for ml. ACM Transactions on Programming Languages and Systems (TOPLAS) 25(1), 117–158 (2003)
19. Schreckling, D., Köstler, J., Schaff, M.: Kynoid: real-time enforcement of fine-grained, user-defined, and data-centric security policies for Android. Information Security technical report 17(3), 71–80 (2013)
20. Yang, J., Yessenov, K., Solar-Lezama, A.: A language for automatically enforcing privacy policies. In: Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principle of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012. pp. 85–96 (2012)