An Automata-based Approach to Evolving Privacy Policies for Social Networks*

Raúl Pardo[1], Christian Colombo[2], Gordon J. Pace[2], and Gerardo Schneider[1]

[1] Department of Computer Science and Engineering,
Chalmers | University of Gothenburg, Sweden.
[2] Department of Computer Science, University of Malta, Malta.
pardo@chalmers.se, christian.colombo@um.edu.mt,
gordon.pace@um.edu.mt, gersch@chalmers.se

Abstract. *Online Social Networks* (OSNs) are ubiquitous, with more than 70% of Internet users being active users of such networking services. This widespread use of OSNs brings with it big threats and challenges, privacy being one of them. Most OSNs today offer a limited set of (static) privacy settings and do not allow for the definition, even less enforcement, of more dynamic privacy policies. In this paper we are concerned with the specification and enforcement of *dynamic* (and *recurrent*) privacy policies that are activated or deactivated by context (events). In particular, we present a novel formalism of *policy automata*, transition systems where privacy policies may be defined per state. We further propose an approach based on runtime verification techniques to define and enforce such policies. We provide a proof-of-concept implementation for the distributed social network Diaspora, using the runtime verification tool LARVA to synthesise enforcement monitors.

1 Introduction

As stated in [21] by Weitzner et al, "[p]rotecting privacy is more challenging than ever due to the proliferation of personal information on the Web and the increasing analytical power available to large institutions (and to everyone else) through Web search engines and other facilities". The problem being not only to determine *who* might be able to access *what* information and *when* but also *how* the information is going to be used (for which *purpose*). Addressing all these privacy-related questions is complex, and as today there is no ultimate solution.

The above is particularly true for *Online Social Networks* (OSNs) (also known as *Social Networking Sites* or *Social Networking Services* —SNSs), due to their explosion in popularity in the last years. Sites like Facebook, Twitter and LinkedIn are in the top 20 most visited Web sites in the world [1]. Nearly 70% of the Internet users are active on OSNs as shown in a recent survey [12], and this number is increasing. A number of studies show that the number of privacy breaches is keeping pace with this growth [15, 10, 14, 16]. The reasons for this increase on privacy breaches are manifold; just to

* Published in the 16th International Conference on Runtime Verification (RV'16), vol. 10012 of LNCS. Springer, 2016.

mention a few: i) Many users are not aware of the implications of content sharing on OSNs, and do not foresee the consequences until it is too late; ii) Most users do not take the time to check/change the default privacy settings, which are usually quite permissive; iii) The privacy settings offered by existing OSNs are limited and are not fine-grained enough to capture desirable privacy policies; iv) Side knowledge and indirect disclosure, e.g. through aggregation of information from different sources, it is difficult to foresee and detect; v) There currently are no good warning mechanisms informing users of the potential breach of privacy, before a given action is taken; vi) Privacy settings are static (they are not time- nor context-dependent), thus not being able to capture the possibility of defining repetitive or recurrent privacy policies.

Recently, the following privacy flaw was pointed out in the Facebook messenger app [3]. It was shown that it is possible to track users based on their previous conversations. It was enough to chat several times per day with users to accurately track their locations and even infer their daily routines. It was possible since the app adds by default the location of the sender to all the messages. This problem arises because of some of the reasons in the previous list such as i), ii) and v). Facebook solution was to disable location sharing by default, which might be seen as a too radical solution. However, it is the best Facebook developers can do given the current state of privacy protection mechanisms. We believe that there is room for better solutions that offer protection to users while not restricting the sharing functionalities of the OSN. For instance, this privacy flaw could have been solved with a privacy policy that says *"My location can only be disclosed 3 times per day"*. This policy prevents tracking users while still allowing users to share their location in a controlled manner. We called this type of privacy policies *evolving* polices and they are the focus of this paper. Other examples of evolving policies are *"Co-workers cannot see my posts while I am not at work, and only family can see my location while I am at home"* or *"My supervisor cannot see my pictures during the weekend"*.

In this paper we address the above problem, through the following contributions: i) The definition of *policy automata* (finite state automata enriched with privacy policies in their states), the definition of a subsumption and a conflict relation between policy automata, and the proofs of some properties about these relations (Section 2); ii) A translation from policy automata into DATEs [4], the underlying data structure of the runtime verification tool LARVA [5] (Section 3); iii) A proof-of-concept implementation of dynamic/recurrent privacy policies for the open source distributed OSN Diaspora* [6] using LARVA (Sections 4,5).

2 Policy automata

In order to describe evolving policies, we adopt the approach of taking a static policy language and use it to describe temporal snapshots of the policies in force. We then use a graph structure to describe how a policy is discarded and another enforced, depending on the events taking place e.g. user actions or system events.

2.1 Semantics of Policy Automata

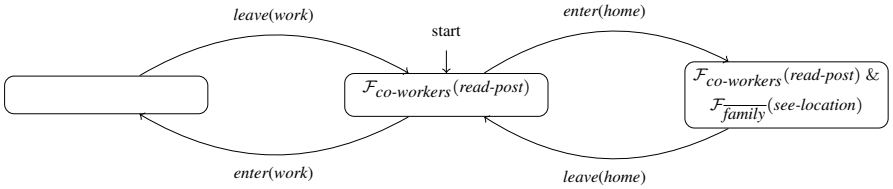
Policy automata are defined as structures such that progressing through structure represents evolving policies, parametrised by a static policy language SPL. This approach allows us to define a whole family of evolving policy languages, depending on the underlying static language used.

Assumption 1 *We assume that SPL has the notion of conjunction of policies such that, for any two policies*[3] $p_1, p_2 \in SPL$, $p_1 \,\&\, p_2 \in SPL$.

Definition 1. *A* policy automaton *over a static privacy policy language SPL is a 4-tuple* $\langle \Sigma, Q, q_0, \rightarrow, \pi \rangle$ *where:* Σ *is the alphabet — effectively the set of observable actions of the underlying system;* Q *is the set of states in the automaton;* $q_0 \in Q$ *is the initial state of the automaton;* $\rightarrow \subseteq Q \times \Sigma \times Q$ *is the transition relation; and* $\pi \in Q \rightarrow SPL$ *is a function which maps each state to a privacy policy in SPL.*

We will write $q \xrightarrow{a} q'$ *to indicate that there is a transition from state* q *to state* q'*, labelled by* a*:* $q \xrightarrow{a} q' \stackrel{df}{=} (q, a, q') \in \rightarrow$*. We will take the transitive closure of the transition relation to enable us to write* $q \stackrel{es}{\Rightarrow} q'$ *to denote that the sequence of events* es *takes the automaton from state* q *to state* q'*.*

Example 1. To illustrate policy automata let us consider the policy *'Co-workers cannot see my posts while I am not at work, and only family can see my location while I am at home'* (P1). If we use the static policy operator $\mathcal{F}_g(x)$ to denote that anyone in group g is forbidden from performing action x (where x can refer to posting, viewing a post, liking a post, etc.), we can express the first part of P1 to be $\mathcal{F}_{\textit{co-workers}}(\textit{read-post})$, and the second part to be $\mathcal{F}_{\overline{\textit{family}}}(\textit{see-location})$ (we use $\bar{g}$ to denote the complement of a group of users g). By synchronising with the actions of our social network application through events marking the arrival at and departure from a location (*enter*(l) and *leave*(l) respectively), we can express the evolving policy in the following manner[4]:



Non-deterministic and non-total transition relations in a policy automaton can lead to policy behaviour which is typically not required in real-life policy analysis. For instance, we do not want to consider automata that under the execution of an event, randomly choose between the activation of two different static policies. For this reason, we define the subset of sane policy automata which behave deterministically and never deadlock.

[3] In the rest of the paper we take SPL to be the set of well-formed policy formulae of the static policy language.

[4] When we draw a policy automaton, transitions for events that are not explicitly drawn are assumed to be reflexive.

Definition 2. *We say that a policy automaton $\mathcal{P} = \langle \Sigma, Q, q_0, \rightarrow, \pi \rangle$ is* sane *if its transition relation is total and deterministic (functional). With sane policies, we write $q \xrightarrow{e}$ and $q \stackrel{es}{\Longrightarrow}$ (with $e \in \Sigma$ and $es \in \Sigma^*$) to denote the unique state reachable from state q, following action e and sequence es respectively. Finally, we will write $policy_{\mathcal{P}}(es)$ to denote the policy in force after following event sequence es from the initial state: $policy_{\mathcal{P}}(es) \stackrel{df}{=} \pi(q_0 \stackrel{es}{\Longrightarrow})$.*

In order to give a semantics to policy automata, we require the semantics of the underlying static policy language. Let $\sigma \in SN$ be the state of the social network where SN is the universe of all possible social network states. Given a static policy language SPL, we write $\sigma,\ e \vdash_{SPL} p$ to denote that in the social network state σ an event e respects privacy policy p. We assume that the social network (but not the policy) may evolve over time through events via the relation $\rightarrow_{SN} \subseteq SN \times \Sigma \times SN$ which is assumed to be a total function on the two first parameters.

Based on the semantics of the static policy language, we can now define the semantics of policy automata:

Definition 3. *The* configuration *of a policy automaton consists of the state of the automaton*[5]*. The initial configuration is taken to be q_0. Whether an* event respects a policy automaton in a particular configuration C *is defined as follows:*

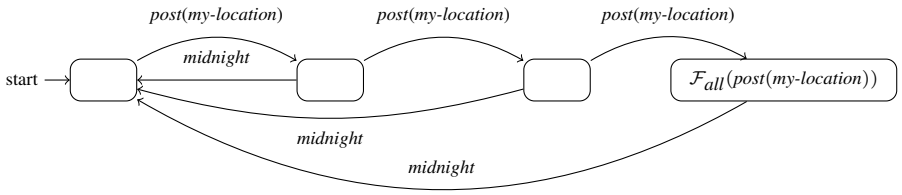
$$\frac{\sigma,\ e \vdash_{SPL} \pi(C)}{\sigma,\ e \vdash_{PA} C}\ \text{SPL}$$

This is extended over traces in the following manner:

$$\frac{}{\sigma,\ \varepsilon \vdash_{PA} C}\ \textsc{BaseTrace}$$

$$\frac{\sigma,\ e \vdash_{PA} C \qquad \sigma \xrightarrow{e}_{SN} \sigma' \qquad C \xrightarrow{e} C' \qquad \sigma',\ es \vdash_{PA} C'}{\sigma,\ e : es \vdash_{PA} C}\ \textsc{IndTrace}$$

Example 2. Consider the policy *'Only up to 3 posts disclosing my location are allowed per day in my timeline'* (P2), which can be encoded as the following automaton (we will assume that from left to right, the states are named q_0, q_1, q_2 and q_3):



[5] We present these semantics in terms of general configurations, rather than the automata states, since we envisage the extension of the automata to handle local symbolic state, requiring a richer configuration but still in line with the definitions given in this paper.

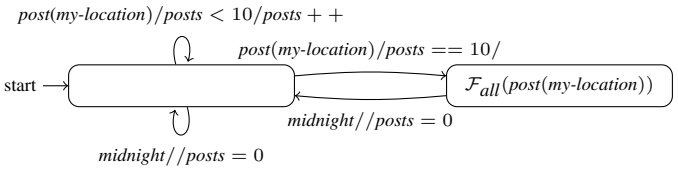
Since we expect that posting the location when a policy prohibiting it is in force is a violation, we would expect the static policy language semantics to show that for any social network state σ: $\sigma,\ \mathit{post}(\mathit{my\text{-}location}) \not\models_{SPL} \mathcal{F}_{\mathit{all}}(\mathit{post}(\mathit{my\text{-}location}))$.

From this, and given that $\pi(q_3) = \mathcal{F}_{\mathit{all}}(\mathit{post}(\mathit{my\text{-}location}))$ we can deduce that in state q_3, the policy clause is likewise violated whenever a post disclosing *my-location* is performed, no matter the state of the social network: $\sigma,\ \mathit{post}(\mathit{my\text{-}location}) \not\models_{PA} q_3$.

Using the rule INDTRACE, provided there is σ' such that $\sigma \xRightarrow{\mathit{post}(\mathit{my\text{-}location})^3} \sigma'$, we have:[6] $\sigma,\ \mathit{post}(\mathit{my\text{-}location})^4 \not\models_{PA} q_0$.

Note that here we write $\mathit{post}(\mathit{my\text{-}location})^4$ because we want to check that after disclosing 3 times the user's location, the forth one would be a violation of $\pi(q_3)$.

If the maximum number of posts were to be increased, the number of states in the automaton would grow quickly. For the sake of presentation, in the rest of the paper, we will also be enriching our notation in the examples to transition systems which have an implicit symbolic state. Transitions are labelled by a triple: *event*/*condition*/*state-update* — triggering when the specified event happens and the condition holds, performing the state update before proceeding. The property allowing for 10 location posts can be expressed in this notation in the following manner:



Such a symbolic automaton can be unfolded into a policy automaton possibly with an infinite number of states. For instance, in the above example, the set of states would be $\{(q, n) \mid q \in \{q_0, q_1\},\ n \in \mathbb{N}\}$ where q holds the value of the (explicit) state, and n the value of *posts*. Since in this paper we are concerned with runtime verification — enforcing a dynamic policy along a single trace, the infinite number of states poses no challenge to the decidability question.

States in policy automata do not contain all the privacy policies which are being enforced in the OSN. Internally the OSN could be enforcing other static policies that have been manually activated by the users. Policy automata are a separate layer to control some static policies. When a policy automaton moves to a state, the static policies in the new state are activated in the OSN. Similarly, when the automaton leaves a state, the static polices are deactivated. Transitions to and from an empty state just mean that there is no update of static policies.

One advantage of using policy automata is that one can combine them synchronously to get the equivalent of conjunction over evolving policies. In order to do so, we require the underlying SPL to have a notion of conjunction (cf. Assumption 1).

Policy automata can now be combined using standard synchronous composition over a particular alphabet:

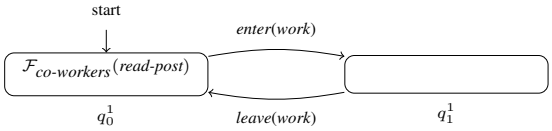
[6] The supra-index over events represent the number of occurrences of the event, so $\mathit{my\text{-}location}^3$ represent the sequence of events *my-location*; *my-location*; *my-location*.

Definition 4. *Given two policy automata $\mathcal{P}_1$ and $\mathcal{P}_2$ (such that $\mathcal{P}_i = \langle \Sigma_i, Q_i, q_{0_i}, \rightarrow_i, \pi_i \rangle$), the synchronous composition of the automata synchronising over actions G, is defined to be the policy automaton $\mathcal{P}_1 \|_G \mathcal{P}_2 = \langle \Sigma_1 \cup \Sigma_2,\ Q_1 \times Q_2,\ (q_{0_1}, q_{0_2}),\ \rightarrow, \pi \rangle$ where $\pi(q_1, q_2) \stackrel{df}{=} \pi_1(q_1)\ \&\ \pi_2(q_2)$ and the transition relation is defined as follows:*

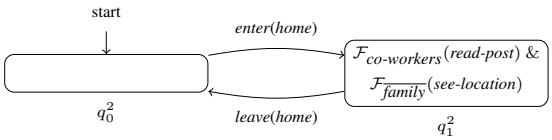
$$\frac{q_1 \xrightarrow{a}_1 q_1' \qquad q_2 \xrightarrow{a}_2 q_2'}{(q_1, q_2) \xrightarrow{a} (q_1', q_2')}\ a \in G$$

$$\frac{q_1 \xrightarrow{a}_1 q_1'}{(q_1, q_2) \xrightarrow{a} (q_1', q_2)}\ a \notin G \qquad \frac{q_2 \xrightarrow{a}_2 q_2'}{(q_1, q_2) \xrightarrow{a} (q_1, q_2')}\ a \notin G$$

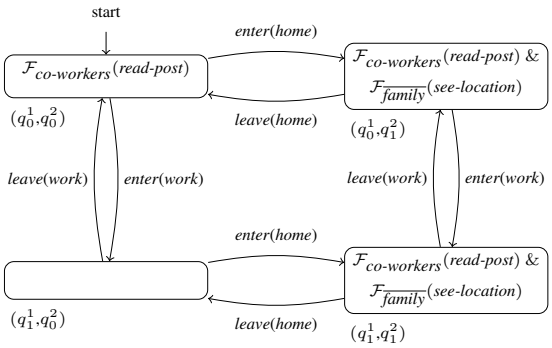
Example 3. The policy automaton of Example 1 effectively is a composition of two individual evolving policies. First *"Colleagues cannot see my posts when I am not at work"*, which can be represented in the following automaton



and secondly, *"Only my family can see my location while I am at home"*:



Let $\mathcal{P}_1$ and $\mathcal{P}_2$ denote the previous two automata, respectively. The following diagram shows $\mathcal{P}_{12}$, the parallel composition of the previous automata $\mathcal{P}_1 \|_\emptyset \mathcal{P}_2$ (the synchronisation set is empty because $\mathcal{P}_1$ and $\mathcal{P}_2$ do not communicate over any event):



Note that this automaton is not equivalent to that of Example 1. In some transitions that Example 1's automaton do not update the static privacy policies (i.e., the automaton remains in the same state) this synchronous composition updates the policies accordingly.

Imagine, for instance, that a user goes from work to home without leaving work (it is a possible scenario if the user lives at her workplace). After receiving *enter(work)*, *enter(home)*, the automaton resulting from the synchronous composition would active the policy $\mathcal{F}_{\textit{co-workers}}(\textit{read-post})$ & $\mathcal{F}_{\overline{\textit{family}}}(\textit{see-location})$ whereas Example 1's automaton would activate no policies. Formally, the state (q_0^1,q_1^2) should contain the static policy $\mathcal{F}_{\textit{co-workers}}(\textit{read-post})$ & $\mathcal{F}_{\textit{co-workers}}(\textit{read-post})$ & $\mathcal{F}_{\overline{\textit{family}}}(\textit{see-location})$. However, we require the & operator of the static policy language to be idempotent (cf. Assumption 2, see below), thus being able to reduce the policy to $\mathcal{F}_{\textit{co-workers}}(\textit{read-post})$ & $\mathcal{F}_{\overline{\textit{family}}}(\textit{see-location})$.

Though formally the evolving policies can thus be combined into a single one, in practice one can keep them separate and enforce them independently, e.g. possibly on separate machines, thus avoiding information leaks (if all the policies) have to be communicated to a central server for enforcement. For instance, one can see a user's set of policies being combined together over his or her local alphabet, and then synchronising globally at a global level across users:

$$(p_{1,1} \|_{U_1} \ldots \|_{U_1} p_{1,n}) \; \|_{\text{Global}} \; (p_{m,1} \|_{U_m} \ldots \|_{U_m} \; p_{m,n'})$$

2.2 Subsumption of dynamic privacy policies

Many notions can be carried over from the underlying static policy language to dynamic policies expressed using policy automata. Provided that the static policy language has a notion of semantic equivalence (which encompasses the usual properties of idempotency, commutativity and associativity of conjunction), we can derive equivalence and strictness ordering over policy automata.

Assumption 2 *We assume that the static policy language SPL has the notion of semantic equivalence $=_{SPL}$ which is assumed to be an equivalence relation.*

Furthermore, conjunction is assumed to be commutative, associative and idempotent under this equivalence: (i) $p_1 \& p_2 =_{SPL} p_2 \& p_1$; (ii) $p_1 \& (p_2 \& p_3) =_{SPL} (p_1 \& p_2) \& p_3$; and (iii) $p \& p =_{SPL} p$.

Based on this equivalence, we can extend this to policy automata equivalence by quantifying over traces:

Definition 5. *Two policy automata $\mathcal{P}_1$ and $\mathcal{P}_2$ (with $\mathcal{P}_i = \langle \Sigma_i, Q_i, q_{0i}, \rightarrow_i, \pi_i \rangle$) with a common alphabet Σ (which requires $\Sigma_1 = \Sigma_2$) are equivalent if after following any trace, they both end up in a state in which the policies are equivalent:*

$$\mathcal{P}_1 =_{PA} \mathcal{P}_2 \stackrel{df}{=} \forall es : \Sigma^* \cdot \textit{policy}_{\mathcal{P}_1}(es) =_{SPL} \textit{policy}_{\mathcal{P}_2}(es)$$

Using standard approach, we can now define policy strictness ordering — a policy is considered stricter than another if all behaviour allowed by the former is also allowed by the latter.

Definition 6. *Given policy automata $\mathcal{P}_1$ and $\mathcal{P}_2$ over alphabet Σ, we say that $\mathcal{P}_1$* is stricter than *$\mathcal{P}_2$, written $\mathcal{P}_1 \sqsubseteq_{PA} \mathcal{P}_2$ as follows:*

$$\mathcal{P}_1 \sqsubseteq_{PA} \mathcal{P}_2 \stackrel{df}{=} \mathcal{P}_1 \|_\Sigma \mathcal{P}_2 =_{PA} \mathcal{P}_1$$

The strictness relation can be shown to obey certain properties.

Lemma 1. *The relation $\sqsubseteq_{PA}$ is transitive, antisymmetric and reflexive.*

Example 4. Consider the policy automaton in Example 1 ($\mathcal{P}_1$) and the synchronous composition of the two automata in Example 3 ($\mathcal{P}_{12}$).

As we remarked in Example 3, the two policy automata are clearly not equivalent. However, we would expect $\mathcal{P}_{12}$ to be a stricter version of $\mathcal{P}_1$. To show this, we note that the synchronous composition of $\mathcal{P}_1$ and $\mathcal{P}_{12}$, $\mathcal{P}_1 \|_\Sigma \mathcal{P}_{12}$ (where Σ is the whole alphabet, including $\{leave(home),\ leave(work),\ enter(home),\ enter(work)\}$), and $\mathcal{P}_{12}$ result in identical policies after following any trace. Formally, for all traces $es \in \Sigma^* \cdot policy_{\mathcal{P}_{12} \| \mathcal{P}_1}(es) =_{SPL} policy_{\mathcal{P}_{12}}(es)$, and thus we can conclude that $\mathcal{P}_{12}$ is stricter than $\mathcal{P}_1$: $\mathcal{P}_{12} \sqsubseteq_{PA} \mathcal{P}_1$.

2.3 Conflicting policy automata

In a similar manner as policy equivalence can be lifted from the static policy language to evolving policies, we can extend the notion of conflicting policies. Two static policies conflict when both cannot be satisfied or enforced at the same time. For example, imagine that Alice sets the policy *"Everyone can see the posts on my timeline"* and Bob activates a policy saying *"Only my friends can see my posts"*. If Bob posts in Alice's timeline which policy would apply? If the audience of the post is only Bob's friends Alice's policy would be violated. Similarly, if the audience of the posts is everyone, Bob's policy would not be satisfied. In order to define conflicting policy automata, we require the static policy language to include the notion of conflict between policies.

Assumption 3 *The static policy language SPL must be equipped with the notion of conflicting polices $\Join_{SPL}$, which is assumed to be (i) symmetric; and (ii) closed under conjunction: if $p_1 \Join_{SPL} p_2$ then for any p_1', it also holds that $(p_1 \ \& \ p_1') \Join_{SPL} p_2$.*

We can lift the static policy conflict relation to one on evolving policies:

Definition 7. *Given any static policy language SPL and policy automata $\mathcal{P}_1$ and $\mathcal{P}_2$ with alphabet Σ:*

$$\mathcal{P}_1 \Join_{PA} \mathcal{P}_2 \stackrel{df}{=} \exists es \in \Sigma^* \cdot policy_{\mathcal{P}_1}(es) \Join_{SPL} policy_{\mathcal{P}_2}(es).$$

The intuition behind the previous definition is simple. Any two automata are in conflict if after the execution of a sequence of events, they end up in a state where their policies conflict (at the static policy level).

Example 5. Imagine that Alice and Bob want to leverage the advantages of evolving policies, and they rewrite the previous static policies in a more precise way, *"Everyone can see the posts on my timeline during my birthday"* and *"Only my friends can see my posts when I am at home"*. Combining the policy automata representing these two policies, we can identify a conflict in a state reachable after a trace in which, Alice's birthday begins and afterwards (before the day ends) Bob goes home. Note that it is not required that Bob posts in Alice's timeline for the conflicting policies to be reached, since it is known beforehand that both policies cannot be satisfied at the same time.

Based on this definition and the assumptions we made about conflicts over static policies, we can prove that evolving policies are closed under increasing strictness.

Theorem 1. *Given the policy automata $\mathcal{P}_1$ and $\mathcal{P}_2$ the following holds*

$$\mathcal{P}_1 \,\text{Ɐ}_{PA}\, \mathcal{P}_2 \;\; \wedge \;\; \mathcal{P}_1' \sqsubseteq_{PA} \mathcal{P}_1 \implies \mathcal{P}_1' \,\text{Ɐ}_{PA}\, \mathcal{P}_2.$$

3 Translation of policy automata to DATEs

Dynamic Automata with Timers and Events (DATEs) [4] are symbolic automata aimed at representing monitors, with a corresponding compilation tool LARVA. In this section, we introduce the basic definitions (leaving out advanced element which are not necessary for this paper) enabling us to provide the translation from policy automata, effectively providing an implementation to the latter through LARVA. As a monitoring formalism, DATE transitions are *event, condition, action* triples: if a matching event occurs and the condition — based on event parameters and the automaton symbolic state — holds, then the action is carried out. The action can be used to either modify the automaton state, interact with the event-generating system, or generate an alert as appropriate.

Definition 8. *A* symbolic automaton *(*SA*) running over a system with state of type Θ, is a quintuple $\langle Q,\ q_0,\ a_0,\ \rightarrow,\ B\rangle$ with set of states Q, initial state $q_0 \in Q$, initial action to be executed $a_0 \in \Theta \to \Theta$, transition relation $\rightarrow \subseteq Q \times \text{event} \times (\Theta \to \mathbb{B}) \times (\Theta \to \Theta) \times Q$, and bad states $B \subseteq Q$. Note that the transitions between automaton states are labelled with: (i) an event expression which triggers the transition; (ii) an enabling condition on the system state — encoded as a function from the system state to a boolean value; and (iii) an action (code) which may change the state of the underlying system — encoded as a function, which given a system state returns an updated system state.*

A total ordering $<$, giving a priority to transitions, is assumed to be given so as to ensure determinism.

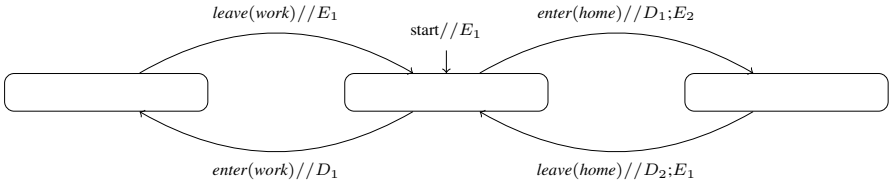
The behaviour of an SA M, upon receiving a set of events, consists of: (i) choosing the enabled transition with the highest priority; (ii) performing the transition (possibly triggering a new set of events); and (iii) repeating until no further events are generated, upon which the automaton waits for a system event.

3.1 Translation

Intuitively, the translation keeps the same states of the policy automaton, but introduces transitions and states for each static policy. We note that the translation below only handles the high-level enabling and disabling of policies, leaving the low-level checking and enforcement up to a static policy checker. We note that the translation below only handles conjunction of policies.

Given a policy automaton $\langle \Sigma, Q, q_0, \rightarrow, \pi \rangle$, for a given transition $(q, e, q') \in \rightarrow$, we generate an action which disables policies in the outgoing state, and enabling those in the ingoing state, as follows: action$(q, e, q') =$ *stopEnforcing*$(\pi(q))$; *startEnforcing*$(\pi(q'))$, where *startEnforcing*(p) and *stopEnforcing*(p) switches on and off the enforcement of static policy p. Using this construction, we generate transitions of the SA labelled as follows: $\rightarrow_{\text{SA}} = \{(q, e, \text{true}, \text{action}(q, e, q'), q') \mid (q, e, q') \in \rightarrow\}$. The resulting DATE would be: $\langle Q, q_0, \text{start}, \rightarrow_{\text{SA}}, \emptyset \rangle$ where start is an action representing the activation of the automaton.

Example 6. Consider the policy automata presented in Example 1, which models the policy *'Co-workers cannot see my posts while I am not at work, and only family can see my location while I am at home'*. Assuming that the events $leave(work)$, $leave(home)$, $enter(work)$ and $enter(home)$ exist, the automaton can be directly converted to a DATE as follows:



where E_1, D_1, E_2 and D_2 are defined as follows:

$$E_1 = \textit{startEnforcing}(\mathcal{F}_{\textit{co-workers}}(\textit{read-post}))$$
$$D_1 = \textit{stopEnforcing}(\mathcal{F}_{\textit{co-workers}}(\textit{read-post}))$$
$$E_2 = \textit{startEnforcing}(\mathcal{F}_{\textit{co-workers}}(\textit{read-post}) \ \&\ \mathcal{F}_{\overline{\textit{family}}}(\textit{see-location}))$$
$$D_2 = \textit{stopEnforcing}(\mathcal{F}_{\textit{co-workers}}(\textit{read-post}) \ \&\ \mathcal{F}_{\overline{\textit{family}}}(\textit{see-location}))$$

4 Implementation in Diaspora* using LARVA

One of our objectives is to have an effective enforcement mechanism for evolving privacy policies based on policy automata in a real OSN. In this section, we describe the details of the implementation of policy automata using LARVA in the OSN Diaspora*.

We chose Diaspora* since it is open source, which allows us to implement the interaction between the OSN and LARVA. Diaspora* has a built-in mechanism for enforcing static privacy policies. Pardo and Schneider have recently extended Diaspora* with a prototype implementation of some privacy policies defined in the $\mathcal{PPF}$ framework [18, 17]. $\mathcal{PPF}$ is a formal (generic) privacy policy framework for OSNs, which needs to be instantiated for each OSN in order to take into account the specificities of the OSN. $\mathcal{PPF}$ was

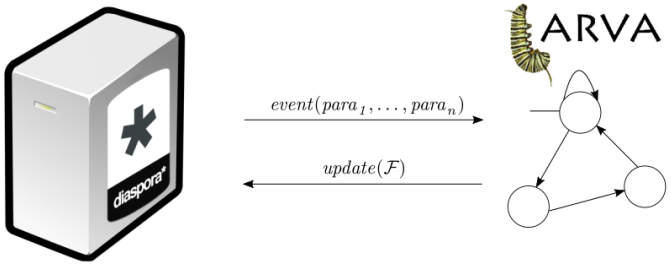


Fig. 1: High-level representation of the Diaspora*-LARVA communication

shown not only to be able to capture all privacy policies of Twitter and Facebook, but also more complex ones involving implicit disclosure of information. $\mathcal{PPF}$ comes with a privacy policy language, $\mathcal{PPL}_{\mathcal{SN}}$, which satisfies all the assumptions placed for the static privacy language in policy automata (cf. Section 2).

Using policy automata to model the evolution of the privacy policies makes it possible to define a modular enforcement of evolving policies. As we mentioned, policy automata are independent of the static policy language of the OSN (except for the assumptions on $=_{SPL}$ and $\&$), and consequently, they are also independent of the underlying enforcement of each particular static policy. Policy automata can be translated to DATEs (cf. Section 3). In order to implement policy automata we use the tool LARVA [5], which automatically generates a monitor from properties expressed in DATEs.

In order for the runtime enforcement to work we use a communication protocol between Diaspora* and LARVA. Every time a relevant event occurs in Diaspora* (i.e., an event that can update the state of the automata), it is reported to LARVA. Then LARVA updates the state of the privacy policies (if applicable), and whenever a privacy policy is updated LARVA reports this change to Diaspora*, which would update the corresponding (static) privacy policy (see Fig. 1).

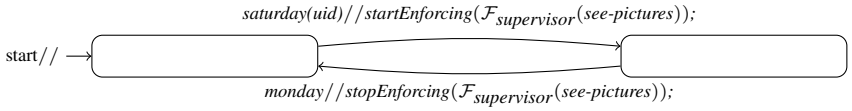
Given that Diaspora* is implemented in Ruby and the monitors that LARVA generates are Java programs, we implement the communication protocol using sockets. One socket is used by Diaspora* to send a message to LARVA containing the event that has occurred, plus additional information such as the user who triggered the event; if it is a post the audience of the post, whether the post contains a location, etc. LARVA monitors detect (among other things) Java method calls corresponding to events on DATE transitions. Therefore, we have implemented a Java program, which listens to the communication socket and depending on the message sent by Diaspora* it calls a concrete method causing the LARVA automaton to update its state. When an automaton updates its state, the privacy policies to be enforced might change. There is another socket that the LARVA monitor uses to send the privacy policies that Diaspora* should enforce. The message sent by the monitor includes the policies that must be activated (policies of the incoming state) and/or deactivated (policies of the outgoing state). This part of the communication will also be handled by the Java program, which contains an auxiliary method for sending messages to Diaspora*.

5 Case studies

As a proof-of-concept we have implemented two policy automata in the Disapora*-LARVA system presented in the previous section. Here we describe the concrete details of this prototype. The code of these case studies can be found in [8].

5.1 Case 1: Protecting pictures during the weekend

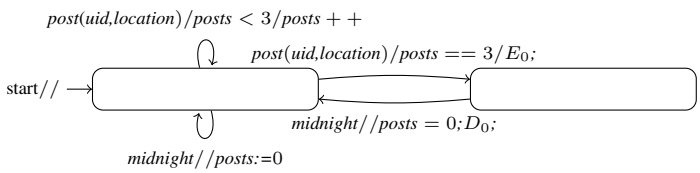
In this case study we describe the implementation of the following evolving privacy policy, *"My supervisor cannot see my pictures during the weekend"*. This is a simple policy that only depends on the time of the week. Let $\mathcal{F}_{supervisor}(\textit{see-pictures})$ represent that my supervisor cannot see my pictures, the following DATE models the policy



As we mentioned, Diaspora*'s privacy protection mechanism is based on an instantiation of $\mathcal{PPF}$. In this instantiation, we consider that a user appears in a picture if the user is mentioned in the post containing the picture[7]. For this policy automaton Diaspora* is required to report the events *saturday* and *monday*. Each of them represents the beginning of the day after which they are named. Every Saturday at 00:00 Diaspora* sends the message `uid;saturday` to LARVA where `uid` is a user id. This message is sent once for each user with her corresponding `uid`. At this point the automaton of each user is updated. The automaton moves to the only possible state where it replies with the message `uid;exclude-supervisor;picture`. When this message is received by Diaspora*, it activates the static privacy policy that forbids posting a picture of a user if her supervisor is part of the audience. More precisely, Diaspora*'s built-in enforcement mechanism will block any post that contains a picture and mention of a user whose supervisor is included in the audience of the post. Similarly, on Monday at 00:00, Diaspora* informs the automata with the message `monday`. All active automata update their state, therefore no `uid` parameter is needed for this event. This choice also reduces the amount of messages sent between Diaspora* and LARVA. Finally, these automata reply to Diaspora* with the message `uid;include-supervisor;picture`, which allows again the user's supervisor to be part of the audience of her pictures.

5.2 Case 2: Disclosing location at most 3 times per day

Here we describe the implementation of the policy automaton of Example 2, which we translate to a DATE (as described in Section 3) as follows



[7] Diaspora* does not support tagging users in pictures.

In the previous automaton $E_0 = \mathit{startEnforcing}(\mathcal{F}_{\mathit{all}}(\mathit{post}(\mathit{uid,location})))$ and $D_0 = \mathit{stopEnforcing}(\mathcal{F}_{\mathit{all}}(\mathit{post}(\mathit{uid,location})))$. Note that we use the variable posts to symbolically encode the explict states of the real policy automata (cf. Section 2). There are two events present in the transitions of the automaton, which therefore need to be reported from Diaspora* to the LARVA monitors when they occur, $\mathit{post}(\mathit{uid,location})$ and *midnight*.

In our Diaspora* $\mathcal{PPF}$ instantiation, mentioning users in a post that includes a location constitutes a disclosure of their location. Every time a user is mentioned in a post (i.e., $\mathit{post}(\mathit{uid,location})$), a message including the message `uid;post;location` is sent to LARVA, specifying the user id and that a location of this user has been disclosed. The message is sent for each user mentioned in the post. As described before, there is one LARVA monitor per user, which controls the policy automaton of each individual. When the message is received the automaton of the user specified by `uid` will be updated. This update will increase the value of the automaton variable posts, whose initial value is 0. After sending the message, Diaspora* waits for the answer of the automaton, in case an update of the privacy policies of the user is required. In case posts is less than 3, there is no need to update the privacy policies, therefore the message `do-nothing` is sent back. On the other hand, if posts is greater than 3, the automaton will move to the state where the policy forbidding the disclosure of locations must be activated, thus it will send the message `disable-posting` to Diaspora*. Note that it is not required to specify the user id in the reply since Diaspora* initiated the communication.

As for the event *midnight*, Diaspora* sends the message `midnight` to the monitors of all users every day at 23:59. If the monitors are in the state where the disclosure of location is forbidden, they take the transition to the initial state. This transition involves, firstly, resetting the variable posts to 0, and secondly, sending the message `uid;enable-posting;location` back to Diaspora*, which removes the privacy policy preventing the location of the user `uid` to be disclosed. If the automaton is already in the initial state, it simply resets posts to 0.

6 Related work

The lack of a temporal dimension in privacy policies was already pointed out by Riesner *et al.* [19]. In their survey, they show that there is no OSN that supports policies that automatically change over time. The authors mention that Facebook allows users to apply a default audience to all their own old posts, but there is a big gap between that privacy policy and the family of evolving policies that we introduce in this paper.

Specifying and reasoning about temporal properties in multi-agent systems using epistemic logic have been the subject of study for a long time. It began with the so called *interpreted systems* (IS). In [7] Fagin *et al.* introduce IS as a model to interpret epistemic formulae with temporal operators such as box and diamond. IS have been used for security analyses of multi-agent systems. Though we do consider a temporal aspect, the focus and objectives of our work are different from the work done in interpreted systems, at least in what concerns the domain of application and the scope of the approach. In our case, the policies themselves are the ones evolving based on events, rather than the information on what is known to different agents at a given time.

Recent research has been carried out in extending IS to be able to reason about past or future knowledge. In [2] Ben-Zvi and Moses extend K_i with a timestamp $K_{i,t}$, making it possible to express properties such as "Alice knows at time 5 that Bob knew p at time 3", i.e., $K_{Alice,5}K_{Bob,3}\ p$. With the same essence but including real time, Woźna and Lomuscio present TCTLKD [22], a combination of epistemic logic, CTL, a deontic modality and real time. In these, and other related work, the intention is to be able to model the time differences in the knowledge acquired by different agents due to delay in communication channels. Although both our motivation as well as the application domain differ from those of the aforementioned logics, it is worth mentioning that they could be indeed useful to express certain real-time policies not currently supported in our formalism.

Despite the richness of both timed epistemic logics, TCTLKD [22] and the epistemic logic with timestamps [2], they would not be able to express recurrent policies as we do. We are of course adding a separate layer beyond the power of the logical formalism by using automata to precisely express when to switch from one policy to another. It remains an interesting question what would be the expressivity of policy automata if we consider an enhancement of $\mathcal{PPF}$ with timed extensions as done in some of the above works in order to express richer (static) policies.

We have not defined here a theory of privacy policies (we have not given a formal definition in terms of traces or predicates), nor have we developed a formal theory of enforcement of privacy policies. To the best of our knowledge such a characterisation does not exist for privacy policies. There is, however, work done in the context of security policies, for instance the work by Le Guernic *et al.* on using automata to monitor and enforce non-interference [11, 9] or by Schneider on security automata [20]. It could be instructive to further develop the theoretical foundations of policy automata and relate it to security automata and their successors (e.g., edit automata [13]).

7 Conclusions

We have presented a novel technique to define and implement evolving privacy policies (i.e., recurrent policies that are (de)activated depending on events) for OSNs. We have defined policy automata as a formalism to express about such policies. Moreover, we have introduced the notion of parallel composition, subsumption and conflict between policy automata and we have proved some of their properties. We have defined a translation from policy automata to DATEs which enables their implementation by means of the tool LARVA. Furthermore, we have describe how to connect LARVA monitors to the OSN Diaspora* so that policy automata can effectively be implemented. In fact, the presented approach would allow to plug in policy automata to any OSN with a built-in enforcement of static privacy policies. Finally, as a proof-of-concept, we have implemented a prototype of two evolving privacy policies.

The policy automata approach has some limitations. For instance, consider that Alice enables the following policy *"Only my friends can see my pictures during the weekend"*. Imagine that Alice and Bob are not friends. If Alice shares a picture on Saturday, Bob will not have access to it. However, on Monday this policy would be deactivated. What would be the effect of turning off this policy? It might be possible that Bob gains access

to all the pictures that Alice posted during the weekend, since no restrictions are specified outside the scope of the weekend. In order to address this problem we might need a policy language able to express *real-time* aspects, with an element of access memory integrated within policy automata.

We are currently also extending policy automata with timing events such as timeouts. This extension will be almost immediately implementable using LARVA since DATEs already support timeouts in their transitions. Another line of work is to extend policy automata with location events. Users normally access OSNs through mobile devices. These devices could directly report the location of users to their policy automata, which avoids having to constantly report users' location to the OSN.

Acknowledgements This research has been supported by: the Swedish funding agency SSF under the grant *Data Driven Secure Business Intelligence*, the Swedish Research Council (*Vetenskapsrådet*) under grant Nr. 2015-04154 (*PolUser: Rich User-Controlled Privacy Policies*), the European ICT COST Action IC1402 (*Runtime Verification beyond Monitoring (ARVI)*), and the University of Malta Research Fund CPSRP07-16.

References

1. Alexa-ranking. http://www.alexa.com/topsites, accessed: 2016-05-11
2. Ben-Zvi, I., Moses, Y.: Agent-time epistemics and coordination. In: Logic and Its Applications, LNCS, vol. 7750, pp. 97–108. Springer (2013)
3. Harvard student loses facebook internship after pointing out privacy flaws. http://www.boston.com/news/nation/2015/08/12/harvard-student-loses-facebook-internship-after-pointing-out-privacy-flaws/, accessed: 2016-05-11
4. Colombo, C., Pace, G.J., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In: 13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'08). LNCS, vol. 5596, pp. 135–149. Springer-Verlag (2009)
5. Colombo, C., Pace, G.J., Schneider, G.: LARVA –A Tool for Runtime Monitoring of Java Programs. In: 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM'09). pp. 33–37. IEEE Computer Society (2009)
6. Diaspora*. https://diasporafoundation.org/, accessed: 2016-05-11
7. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about knowledge, vol. 4. MIT press Cambridge (2003)
8. $\mathcal{PPF}$ Diaspora*. Test pod: https://ppf-diaspora.raulpardo.org. Code: https://github.com/raulpardo/ppf-diaspora, 2016
9. Guernic, G.L.: Automaton-based confidentiality monitoring of concurrent programs. In: 20th IEEE Computer Security Foundations Symposium (CSF'07). pp. 218–232 (2007)
10. Johnson, M., Egelman, S., Bellovin, S.M.: Facebook and privacy: It's complicated. In: Proceedings of the Eighth Symposium on Usable Privacy and Security. pp. 9:1–9:15. SOUPS '12, ACM, New York, NY, USA (2012)
11. Le Guernic, G., Banerjee, A., Jensen, T., Schmidt, D.A.: Automata-Based Confidentiality Monitoring, pp. 75–89. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
12. Lenhart, A., Purcell, K., Smith, A., Zickuhr, K.: Social media & mobile internet use among teens and young adults. Pew Internet & American Life Project (2010)
13. Ligatti, J., Bauer, L., Walker, D.: Edit automata: Enforcement mechanisms for run-time security policies. International Journal of Information Security 4, 2–16 (2005)

14. Liu, Y., Gummadi, K.P., Krishnamurthy, B., Mislove, A.: Analyzing facebook privacy settings: User expectations vs. reality. In: Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference. pp. 61–70. IMC '11, ACM (2011)
15. Madejski, M., Johnson, M., Bellovin, S.: A study of privacy settings errors in an online social network. In: IEEE International Conference on Pervasive Computing and Communication Workshops. pp. 340–345. (PERCOM Workshops'12) (2012)
16. Madejski, M., Johnson, M.L., Bellovin, S.M.: The failure of online social network privacy settings. Columbia University Computer Science Technical Reports (2011)
17. Pardo, R.: Formalising Privacy Policies for Social Networks. Department of Computer Science and Engineering, Chalmers University of Technology (2015), pages 102. Licentiate thesis.
18. Pardo, R., Schneider, G.: A formal privacy policy framework for social networks. In: SEFM'14. LNCS, vol. 8702, pp. 378–392. Springer (2014)
19. Riesner, M., Netter, M., Pernul, G.: An analysis of implemented and desirable settings for identity management on social networking sites. In: Availability, Reliability and Security (ARES), 2012 Seventh International Conference on. pp. 103–112 (Aug 2012)
20. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. 3(1), 30–50 (2000)
21. Weitzner, D.J., Abelson, H., Berners-Lee, T., Feigenbaum, J., Hendler, J.A., Sussman, G.J.: Information accountability. Communincations of the ACM 51(6), 82–87 (2008)
22. Woźna, B., Lomuscio, A.: A logic for knowledge, correctness, and real time. In: Computational Logic in Multi-Agent Systems, LNCS, vol. 3487, pp. 1–15. Springer (2004)