Compliance Monitoring of Third-Party Applications in Online Social Networks

Florian Kelbert
Technische Universität München
Garching b. München, Germany
kelbert@in.tum.de

Alexander Fromm
Technische Universität München
Garching b. München, Germany
fromm@in.tum.de

***Abstract*—With the widespread adoption of Online Social Networks (OSNs), users increasingly also use corresponding third-party applications (TPAs), such as social games and applications for collaboration. To improve their social experience, TPAs access users' personal data via an API provided by the OSN. Applications are then expected to comply with certain security and privacy policies when handling the users' data. However, in practice, they might store, use, and distribute that data in all kinds of unapproved ways. We present an approach that transparently enforces security and privacy policies on TPAs that integrate with OSNs. To this end, we integrate concepts and implementations from the research areas of data usage control and information flow control. We instantiate these results in the context of TPAs in OSNs in order to enforce compliance with security and privacy policies that are provided by the OSN operator. We perform a preliminary evaluation of our approach on the basis of a TPA that integrates with the Facebook API.**

I. Introduction

Today, billions of users interact using Online Social Networks (OSNs) such as Facebook, Twitter, and LinkedIn. These users increasingly use OSNs' third-party applications (TPAs) that offer additional services, such as social games and entertainment, but also productivity apps such as slide sharing. To provide their social experience, TPAs ask for their users' consent to retrieve data about them from the OSN, e.g. personal information, photos, messages, and contact lists. TPAs then access the users' data from the OSN via well-defined APIs. Noteworthy, OSNs use OAuth [4] to ensure that TPAs only access data for which the corresponding users gave their explicit consent.

Yet, once a TPA retrieved user data, it remains unclear whether, how and where this data is stored, processed, and disseminated [3, 8, 15, 20]. Noteworthy, TPAs that integrate into major OSNs are obliged to present a privacy policy and terms of service to users. In addition, many OSN operators demand TPA providers to adhere to certain security and privacy policies [6, 17, 24]. Laws and regulations further constrain the allowed data usage. Facebook even performs a manual security audit of TPAs in case they request access to certain sensitive data. Technically, however, there do not exist means for users, OSN operators, and legislators to monitor whether TPAs actually comply with the above policies and regulations. Even if the application is audited before its initial deployment, the application developers might change its behavior soon thereafter, possibly using data in illegitimate ways.

We provide an OSN-independent solution to enforce compliance of TPAs with security and privacy policies, such as "you may cache the content for up to 24 hours" [17], "Only use friend data in the person's experience in your app" [6], and "You may not disclose confidential information to a third party without the prior explicit consent of Tumblr." [24]. To do so, we integrate concepts and technologies from the research field of *data usage control* and tailor them to the particular needs of online social networks and their TPAs. While most parts of our infrastructure and implementation generalize to scenarios beyond OSNs, we provide a prototype for Facebook.

Our solution works as follows: Data usage control technology [7, 12, 18, 19] is integrated into the Software Execution Environment (SEE) (e.g., Java Runtime Environment (RTE), .NET CLR, PHP interpreter) which is then supposed to be provided in a Platform-as-a-Service (PaaS) manner by trustworthy entities. Such trustworthy entities might include the OSN operator herself, public universities, or consumer protection organizations (cf. Assumptions). TPA providers will then choose to deploy their application on these platforms, the incentive being to provide concerned users technical confidence in their application's compliance with security and privacy policies. Once the TPA is accessed by the user and retrieves her personal data from the OSN, the data is accompanied with machine-enforceable data usage policies. Compliance with these policies is then transparently enforced by the underlying usage control enabled SEE.

We prototypically implement the described concepts for the Java RTE, which will then serve as a usage control enabled SEE. TPAs are implemented in Java and deployed as Java Web Application Archives. Once any such TPA is deployed on a data usage controlled SEE and retrieves user data from the OSN, the application's compliance with selected policies from the Facebook Platform Policy [6] is transparently enforced.

Example Use Case. Alice uses *BirthdayCalendar*, an application that displays the dates of birth of her Facebook friends. After Alice logs in, BirthdayCalendar (i) fetches Alice's friendlist from Facebook, (ii) fetches the dates of birth of all of her friends, and (iii) renders these dates as a calendar within her browser. Using existing technology, BirthdayCalendar might use the received data in any imaginable way. However, if BirthdayCalendar is run on a PaaS which uses

a usage control enabled SEE, then BirthdayCalendar can be kept from data misuse. We refer to this example throughout the paper. Note that we use BirthdayCalendar just for the sake of illustration; Facebook provides a similar built-in function.

Problem. Many legal contracts and regulations define how data may be stored, processed, and disseminated by TPAs in the context of OSNs. However, we are not aware of technical means that enforce compliance.

Solution. We leverage that web applications are increasingly executed in the cloud and show how data usage control technology can be integrated into SEEs on which modern PaaS platforms are based. We assume PaaS platforms to be operated by trustworthy entities. Providers of TPAs are supposed to run their applications on these compliance-monitoring platforms, thus assuring users that their data is not misused.

Contribution. We integrate several methodological and technical data usage control solutions and instantiate them to the ecosystem of OSNs and their TPAs. To the best of our knowledge, we are the first to generically and transparently ensure the compliance of TPAs with security and privacy policies, thus increasing users' confidence when using TPAs.

Assumptions. We consider the OSN operator to be trustworthy w.r.t. the proper processing, storage, and dissemination of user data. We believe that this is a realistic assumption, since it is in the very interest of the operator that user data is used in accordance with laws, terms and conditions, and the users' preferences. Violations would result in reputational damage. In fact, users implicitly trust OSN operators, as they intentionally disclose their information on the corresponding platforms. Since untrustworthy TPAs may have severe impact on the OSN's reputation, the OSN operator seeks their compliance.

Another central assumptions is that the PaaS operator is trusted. We believe that this assumption is realistic, as the OSN operator, which is entrusted with the users' data in the first place, could provide a PaaS. Similarly, organizations that are generally trusted by the public, such as customer protection organizations or universities, could act as PaaS provider.

Attacker Model. We focus on misbehaving TPAs that try to process, store, or disseminate user data without complying to respective policies. Note that such behavior might be deliberate or accidental. Social plugin buttons (*Like*, *+1*) are out of scope. OSN end users are not considered attackers, as previous work showed how they can be kept from data misuse.

Generalization of the Approach. The work at hand integrates multiple previous works and instantiates them to one particular use case. Our approach is generic in that it is independent of the OSN, the TPA, as well as the policies. Beyond the current scenario, we are confident that both our general approach as well as our implementation can be easily ported to mobile phones, in which applications request sensitive user data from the operating system. Generalizing even further, the underlying data usage control solutions and technologies generalize to any scenario in which privacy policies ought to be enforced.

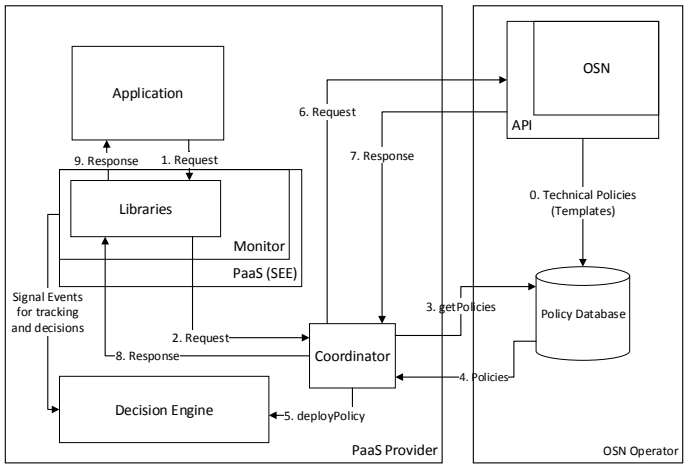

Fig. 1. Crucial components and their interactions as detailed in § II.

II. Approach

After giving an overview over our approach in § II-A (Fig. 1), the subsequent sections detail the involved entities, their functionalities, and their interactions. This section is intentionally kept high-level; details are provided in § III.

A. Overview

OSNs provide an API which allows TPAs to access users' personal data, such as photos, contacts, email addresses, etc. OSN operators protect access to this data using OAuth [4]. Hence users must give their explicit consent to release such data to TPAs. However, once TPAs gained access to such personal data, there do not exist technical means to enforce their compliance with security and privacy policies which are provided by the OSN operator in natural language [6, 17, 24].

In order to foster TPAs' compliance with such policies, we assume that each OSN operator provides a *Policy Database* containing machine-readable policies in line with aforementioned natural language policies. Leveraging the trend to run applications on cloud platforms, we further suppose the providers of TPAs to deploy and run their applications in a PaaS-manner on platforms which are operated by trusted parties. Technical monitors are then integrated into the SEE (i.e., the *PaaS* platform), allowing them to enforce the compliance of TPAs with provided policies in a transparent manner, i.e. without the need to adapt existing applications.

While the monitors integrated into the PaaS take care about policy enforcement, they depend on an external *Decision Engine* from the literature [12, 19] which is responsible for decision making and further management tasks. In order to keep the architecture modular, a Coordinator coordinates the interactions between components. The following sections detail the general workflow and the involved components.

B. Policy Provisioning via the Policy Database

The security and privacy policies to be enforced on TPAs are usually not available in a machine-readable format but rather in natural language. Hence, a first step towards their automated

enforcement is their translation into appropriate formats. Since our enforcement infrastructure builds upon data usage control solutions, we demand policies to be formalized as Event-Condition-Action (ECA) rules. Thereby, conditions are written in an *Linear Temporal Logic (LTL)* dialect as detailed in [14, 19], which allows to formalize propositional, temporal, and cardinal constraints. Actions of ECA rules include inhibition and allowance of the attempted event, its execution in a modified manner, as well as the execution of additional events. In the following, we derive an ECA rule in an intuitive manner. Structured and semi-automated derivation of such ECA rules from natural language policies is possible [16].

Facebook's policy "If you cache data you receive from us, [...] keep it up to date" [6] may be interpreted as follows: *Whenever some data $data$ is processed by the application, then it must have been received from Facebook within the last 24 hours*. Using the formalism described in [11, 14, 16, 19], one possible ECA rule expressing this policy is:

$$\begin{aligned} \text{Event:} &\quad process(data) \\ \text{Condition:} &\quad not(repmin(24[hours], 1, receive(data)) \\ \text{Action:} &\quad inhibit \end{aligned}$$

The semantics are as follows: If an attempt to execute event $process(data)$ is observed *and* if the actual execution of that event would make the condition $not(repmin(24[hours], 1, receive(data))$ true, i.e. if the data being processed was *not* received ($receive(data)$) *at least once* from Facebook within the last *24 hours* ($repmin(24[hours], 1, \ldots)$, then the attempted event is to be inhibited. Note that this ECA rule is actually an *ECA rule template* which will be instantiated for different types of data as required. This step is detailed in § II-D.

We assume the OSN operator to take care of the above policy translation process and the provisioning of the resulting ECA rule templates via a Policy Database (Fig. 1, step 0). This ensures that different PaaS operators do not interpret the same high level policy differently. Note that the above ECA rule template is abstract, as it depends on implementation details what it means to $process$ or $receive$ data. We instantiate the above ECA rule template to one specific SEE in § III.

C. PaaS Operation

We hypothesize that developers of TPAs are willing to run their application on a PaaS hosted by a trusted operator, thus indicating to users that the application handles user data in compliance with provided policies. Our approach enables PaaS operators to enforce ECA rules on any TPA as follows.

In order to ease control over the deployed applications, the PaaS is configured to only allow access to the OSN's API via well-defined libraries. Further, the PaaS' SEE is enriched with monitoring technology which allows to (i) intercept events happening within deployed applications (e.g., $process(data)$, $receive(data)$, $transfer(data)$), (ii) notify those events to the Decision Engine, (iii) enforce the Decision Engine's decision. How interception and notification of events, as well as enforcement of the decision is technically achieved differs between SEEs; one instantiation is described in § III. The following paragraphs abstractly describe the general workflow.

Once a user accesses a TPA for the first time, she is required to grant the application access to certain parts of her user data being stored at the OSN. Note that granting such access is a prerequisite for the usage of the TPA, otherwise the OSN would deny access and not return any user data; major OSNs use the OAuth protocol for this purpose.

Whenever a TPA requests user data from an OSN via any of the libraries provided by the PaaS (Fig. 1, step 1), the PaaS redirects these requests to the Coordinator (Fig. 1, step 2). The Coordinator inspects the request in order to determine which *types* of data (e.g. date of birth, contacts, email address, photos) are requested from which OSN. This is important because different policies, i.e. ECA rule templates, might be applicable for different data types. Hence, the Coordinator retrieves all applicable ECA rule templates from the OSN's Policy Database (Fig. 1, steps 3/4) and deploys them at the Decision Engine (Fig. 1, step 5) as further detailed in §§ II-D and III-B. Once the successful deployment was acknowledged, the Coordinator forwards the original request to the OSN (Fig. 1, step 6). Upon retrieval of the corresponding response (Fig. 1, step 7), it merely remains to forward the payload (actual data) to the requesting application (Fig. 1, steps 8/9).

D. Policy Instantiation and Deployment

Before the Coordinator may deploy an ECA rule at the Decision Engine, the corresponding ECA rule template received from the Policy Database must be instantiated for the concrete data being protected. For this, the Coordinator replaces all placeholders within the template with concrete values. For the above template, $data$ is replaced with a uniquely generated id, say $d9235$, which from now on identifies the data received from the OSN for its entire lifetime:

$$\begin{aligned} \text{Event:} &\quad process(d9235) \\ \text{Condition:} &\quad not(repmin(24[hours], 1, receive(d9235)) \\ \text{Action:} &\quad inhibit \end{aligned}$$

The Coordinator then (i) sends this ECA rule to the Decision Engine for actual enforcement and (ii) informs the Decision Engine about the presence of data $d9235$ and its initial storage location (*container*), say variable c_1, within the TPA. This container reflects where the requested data will be stored by the TPA and might be a file, a memory address, a variable, etc. This differentiation between an abstract data id ($d9235$) and the data's actual storage location(s) (e.g., c_1) is necessary to be able to enforce ECA rules not only on particular copies of that data, but on *all* of them. Note how the above ECA rule talks about the protection of one such abstract data id (i.e., $d9235$) rather than concrete containers (e.g., c_1). In order to enforce the ECA rule on all copies of the data, the flow of this protected data (identified by $d9235$) throughout the TPA, and possibly beyond, must be tracked.

E. Data Flow Tracking

The Decision Engine tracks data flows within TPAs that are deployed on the PaaS. For this, it maintains a mapping

between the data identifiers being tracked and these data's containers at each point in time. Considering the above example of informing the Decision Engine that data $d9235$ is in container c_1, this implies a mapping from c_1 to $d9235$.

In order for this mapping to accurately reflect the data's actual distribution within the TPAs at each point in time, the SEE informs the Decision Engine about any relevant events, such as copying data from one variable to another, sending data to a network socket, or writing data to a file. Whenever such an event is signaled to the Decision Engine, the latter updates its mapping in correspondence with the event's semantics. For example, if the event $copy(c_1, c_2)$ copies all content from container c_1 to container c_2, then c_2 is updated to contain all data that is associated with c_1. Such event semantics were defined and implemented for a multitude of applications and platforms such as MS Windows [25], Unix [10], and Java [7].

F. Policy Decisions and Decision Enforcement

Whenever the SEE observes an *attempt* to execute an event, it must be determined whether this event is in compliance with the deployed policies. For this, the SEE signals any such event to the Decision Engine *before execution*. Based on the history of events, the current mappings between data and containers, as well as the deployed policies, the Decision Engine decides about the event's compliance. While this decision process is detailed in [12, 14, 19], we describe its most relevant aspects.

Assume that a TPA stored data in a variable c_3 and performs computations on that variable. Then, depending on the implementation, the SEE might observe this computation and signal event $process(c_3)$ to the Decision Engine in order to get a decision whether this event is compliant. Notably, however, the deployed policies are formulated in terms of data ids ($d9235$), whereas events observed by the SEE always refer to containers (c_3). Thus, in order to take a decision, the Decision Engine uses its mapping between containers and data ids to know whether at this particular point in time container c_3 contains any data id for which a policy was deployed (such as $d9235$). In the light of this information, the corresponding policies are evaluated and a corresponding decision is taken and sent to the SEE for enforcement.

The Decision Engine might take the following decisions: *Allowance* indicates that the signaled event does comply with all deployed policies and hence it may be executed. In other words, there was no ECA rule for which the signaled event matched the trigger event *and* for which the condition evaluated to *true*. If however, the signaled event matched an ECA rule's trigger event *and* if that event made the ECA rule's condition *true*, then the Decision Engine's decision depends on the matched ECA rule's action: *Inhibition* implies that the SEE must suppress the signaled event's execution, thus preventing non-compliant behavior of the TPA. An alternative would be to allow the *modified execution* of the signaled event, e.g. by demanding anonymization of parts of the processed data. In this case the SEE might enforce the policy by replacing certain parts of the data with Null-Bytes. Lastly, the ECA rule's action might specify to allow/deny the event in question, but demand the execution of additional events, such as logging or notification of an administrator. While in this last example TPAs might behave in a non-compliant manner, it is possible to detect such behaviour a posteriori.

III. Implementation

To show the feasibility of our approach, we (i) translated real-world natural language Facebook Platform Policies into ECA rule templates and provide them in XML format via a Policy Database, (ii) implemented a Monitor for the Java Virtual Machine which is used to monitor Java Web Services that integrate with the Facebook Graph API, (iii) implemented an example BirthdayCalendar web service that integrates with the Facebook Graph API, (iv) implemented a generic Coordinator which is independent of the PaaS, online social network, and policies being used, and (v) instantiated all of the above together with an existing Decision Engine.

A. Application Deployment and Monitoring

Whenever a TPA is deployed on the PaaS, its compliance with the OSN's policies must be assured. Our prototype provides such compliance assurance for Java-based web applications, i.e. Java Web Application Archives (WARs).

We leverage that TPAs exhibit data input channels (*sources*) and data output channels (*sinks*), through which data enters and leaves the application, respectively. Examples for such channels are (parameters of) methods for reading/writing from/to files or sockets. In between the execution of such channels, a multitude of instructions implement the application's functionality. Importantly, those instructions define data flow dependencies between sources and sinks. We enforce compliance with policies by (i) monitoring which protected data enters the application via sources, (ii) tracking this data through the application, and (iii) inhibiting or modifying the execution of sinks if this would violate policies. Technically, our approach is *hybrid* by performing both a *static analysis* of the application and *dynamic runtime tracking* as follows.

Static analysis. Upon deployment of the TPA's bytecode at the PaaS, we perform a static analysis on the application in order to detect all sources and sinks as well as possible data flow dependencies between them. Sources and sinks that target the OSN's API are of particular interest. As we assume the PaaS to only allow access to the OSN's API via provided libraries (cf. § II-C; our implementation leverages RestFB[1]), the PaaS operator specifies in an application-independent manner which kinds of sources and sinks (i.e., which method parameters) of which libraries should be considered for the static analysis. We then use Joana [9], a static information flow analysis tool for Java-based applications, to retrieve a static report of sources, sinks, and potential data flow dependencies, cf. Fig. 2. For each source and sink the report provides the method and bytecode-offset (`Location`), the method signature of the sink/source (`Signature`), as well as which parameters of the method invocation are classified as sink/source (`Return`

[1] http://www.restfb.com

```
Sources:
  Source36:
    Location: Common.updateCurrentUser(...)V:61
    Signature: com.restfb.DefaultFacebookClient.
             fetchObject(Ljava/lang/String;
             Ljava/lang/Class;[Lcom/restfb/Parameter;
         )Ljava/lang/Object;
    Return

Sinks:
  Sink15:
    Location: Main.loggedInWork(...)V:153
    Signature: org.apache.catalina.connector.
             CoyoteWriter.println(Ljava/lang/String;)V
    ParamIndex: 1

Flows:
  Sink15 --> Source36
```

Fig. 2. Abstracted Joana report of sources, sinks, and dependencies (flows).

or `ParamIndex`). In addition, dependencies between sources and sinks are reported (`Flows`).

Dynamic runtime tracking. In contrast to pure dynamic data flow tracking, in which each instruction is monitored, we leverage Joana's results and inject additional instructions selectively only at those locations within the TPA that are relevant for tracking data flows and enforcing policy compliance.

We use the *OW2-ASM*[2] framework to inject additional instructions for each reported flow into the bytecode of the TPA. At runtime, those instructions extract context information (e.g., what kind of source/sink was executed) and signals corresponding events to the Decision Engine in order to (i) verify if the execution of the sink/source is compliant with the deployed policies, and (ii) to track the flow of data from sources to sinks.

Whenever a source is executed, the injected instructions signal this fact to the Decision Engine, thus reporting that sensitive data was read by the TPA. The Decision Engine leverages this information for later decision making: Once a sink is attempted to be executed, the injected instructions signal this mere execution attempt to the Decision Engine in order to verify if the actual execution of the sink is compliant with the deployed policies. For taking this decision, the Decision Engine not only considers this current information and the policies, but also information about sources that were executed earlier as well as the reported information about data flow dependencies. In case the attempted execution of the sink is compliant, the Decision Engine records this fact of execution and propagates the flow of data from the sources to the sink in correspondence with the statically computed data flow dependencies. In case execution of the sink is not compliant, the Decision Engine replies with *Inhibition* or *Modification* which will then be enforced by the injected instructions.

B. Requests for User Data and Policy Deployment

Any request from the TPA to the Facebook Graph API is redirected to the Coordinator (cf. § II-C). We realize this redirection by modifying a few lines in classes `java.net.InetAddress` and `java.net.URL` of OpenJDK8, which is used to run the JRE-based SEE. The additional code essentially checks whether the contacted hostname refers to a supported OSN (e.g., Facebook, Google+), and, if so, subsitutes that hostname with that of the Coordinator; the original request is consequently redirected. Further, one additional URL GET parameter (`osn`) is added to the request in order to keep track of the online social network for which the request was intended. For example, the request ⟨`https://graph.facebook.com/me/?fields=inbox`⟩, which requests the current user's message inbox, is modified to ⟨`http://coordinator/me/?fields=inbox&osn=FB`⟩. The substituting host `coordinator` is configurable. Parameter `osn=FB` indicates to the Coordinator that the request was intended for Facebook—information that was originally encoded within the URL. Note that the above modification replaces HTTPS by HTTP. Since the Coordinator is within the domain of the PaaS provider, we do not consider this to be insecure. Notably, the communication between the Coordinator and the OSN does use HTTPS.

Upon receiving such a request, the Coordinator extracts the requested data types (for Facebook: URL parameter `fields`) as well as the OSN for which the request was intended (parameter `osn`). Using this information, the Coordinator retrieves all applicable policy templates from the Policy Database (§ II-B), instantiates them (§ II-D), and deploys the resulting ECA rules at the Decision Engine.

In accordance with the ECA rule specification in § II-B, Fig. 3 shows a concrete XML policy template specifying that processed data must not be older than 24 hours. Note that placeholders `{DATA_ID}` and `{CONTAINER_NAME}` are filled when the policy is instantiated. For each `mechanism`, the `trigger` specifies upon which action the mechanism is applied. Concretely, this is the case whenever the trigger action, all parameter names, as well as all values match a signaled event. For instance, the trigger action in Fig. 3 specifies that the mechanism is triggered whenever a source event is attempted (`attempt="true"`) that selects (`calleeMethod="execute"`, `sqlType="SELECT"`) data from a database (`calleeClass="java.sql.Statement"`) where the value of `identifier` is part of the query-statement. The `condition` specifies the ECA rule's condition. Concretely, the condition formalizes that within the last 24 hours there must have occurred at least one (`limit="1"`) "Source" event that retrieved the data in question from the Facebook-API. The `action` specifies which actions are executed once the trigger event occurs and the condition evaluates to `true`. Here, the triggering event's parameters are overwritten with default values.

Before deployment, the Coordinator replaces the placeholder values with concrete values. For doing so, we assign each source a unique identifier and signal this identifier to the Coordinator whenever a request to the OSN's API is made. Once the Coordinator receives such an identifier, it replaces the placeholder within the policy template and deploys the resulting ECA rule at the Decision Engine. Upon

[2]http://asm.ow2.org

```
<policy>
  <mechanism>
    <trigger action="Source" attempt="true">
      <param name="calleeClass" val="java.sql.Statement" />
      <param name="calleeMethod" val="execute" />
      <param name="sqlType" val="SELECT" />
      <param name="identifier" val="{DATA_ID}" />
    </trigger>

    <condition>
      <repmin amount="24" unit="HOURS" limit="1">
        <event action="Source" tryEvent="false">
          <param name="calleeClass"
            val="com.restfb.DefaultFacebookClient"/>
          <param name="calleeMethod" val="fetchObject" />
          <param name="identifier" val="{DATA_ID}" />
        </event>
      </repmin>
    </condition>

    <action>
      <allow>
        <modify>
          <param name="firstname" val="John"/>
          <param name="lastname" val="Doe"/>
          <param name="gender" val="unknown"/>
        </modify>
      </allow>
    </action>
  </mechanism>

  <initialRepresentations>
    <container name="{CONTAINER_NAME}">
      <dataId>{DATA_ID}</dataId>
    </container>
  </initialRepresentations>
</policy>
```

Fig. 3. Policy template specifying to not process data older than 24 hours.

policy deployment, the Decision Engine analyzes the policy's `initialRepresentation`-tag and creates a container that serves as initial storage location for the data to be received from the OSN.

IV. Preliminary Performance Evaluation

As we inject additional instructions into the application, we performed a preliminary evaluation of the imposed runtime overhead along our motivating example from § I.

Experiment Setup. We run our experiments on a virtual machine with 8GB RAM and a 4-core 2.6GHz CPU. We instantiated a simplified PaaS platform, i.e. a single instance of *Apache Tomcat 8*, which was equipped with our runtime monitor and provided the SEE for running BirthdayCalendar. The server provided the `restfb`-library, which is used by BirthdayCalendar to query Facebook's API. The Decision Engine and the Coordinator run on the same machine.

The BirthdayCalendar provides a `Login`-function that authenticates a user at Facebook, and a `Rendering`-function that fetches and displays the dates of birth of the authenticated user's Facebook friends. Combining these functionalities, we executed two sequences of actions, 50 times each: (`A1`) `Login` and `Rendering`, and just (`A2`) `Rendering`. We measured the native and instrumented execution times of our prototype.

Results. Native and instrumented average runtimes for `A1` and `A2` are reported in Table I. We observe that the relative overhead ($Relative = Instr./Native * 100 - 100$) imposed by action `A1` is comparatively small (15%) in contrast to `A2` (41%). The reason is that `A1` queries the Facebook API twice: Once to authenticate the user, and once to fetch the user's data, i.e. the friendlist and all corresponding dates of birth. In contrast, `A2` queries the Facebook API only once, since the user is already authenticated. Based on these early results, it seems that our infrastructure exhibits a lower relative performance overhead for computationally expensive tasks for which even native runtimes are high (`A1`, authentication).

TABLE I
Average runtime performance in milliseconds.

Action	Native	Instr.	Relative
`A1`	394.60	454.28	15%
`A2`	49.89	70.50	41%

To identify possible options for performance optimization, we also measured the time that it takes to craft an event-signalling message that is to be sent to the Decision Engine for decision taking purposes (`Create`). Further, we measured the time that it takes to signal this event to the Decision Engine, have a decision taken, and have the decision signaled back (`Signal`). We realize that the event creation time `Create` averages at 13.95ms, while the event signaling time `Signal` averages at 3.5ms. To improve runtime performance, we thus plan to provide tailored event templates that can be quickly instantiated with concrete values at runtime. While the event signaling time `Signal` is comparatively low, signaling events to the Decision Engine via function calls rather than socket communication would definitely contribute to even lower runtimes. Admittedly, our current experiment setup does not cater to potential network latencies in case the infrastructure's components are deployed remotely from one another.

We expect to significantly improve upon the above numbers by integrating the suggested improvements as future work.

V. Limitations

Static Information Flow Analysis. By using Joana for static analyses of TPAs, our prototype does not support applications that make use of callback handlers. Indeed, this is a challenge for any static program analysis, as additional knowledge on the usage of callback handlers would be required. Such knowledge could be obtained by executing the application, by specifying it in a dedicated language, or by simulating all possible callbacks. Java reflections pose a similar challenge. Additional code analyses might reveal reflective code, but generally either coarse assumptions are required or unresolvable reflective code has to be ignored. Bodden et al. [2] exploit runtime information to resolve reflective code.

Java Native Interface (JNI). Java applications may ship together with native libraries which are then used via JNI to perform native operations, e.g. on files or the network. JNI might thus be used to provide custom implementations to communicate over the network, e.g. by mimicking the behaviour of `DatagramSocket.send()`. We assume the PaaS operator

to disallow the usage of such native code, e.g. by scanning the deployed applications for native libraries and rejecting them.

System Boundaries. We enforce policies at and above the PaaS layer. Once data is about to leave these system boundaries, e.g. if data is sent to a remote system, the infrastructure must either ensure that remote systems are equipped with a similar data protection infrastructure [12, 13], or that sensitive data is not disclosed at all. Similarly, our approach is not able to prevent data misuse at software or hardware layers below the PaaS. To mitigate corresponding attacks, usage control technology would need to be implemented at these layers, either at the operating system layer [10, 25] or below.

Enforcement of Policies. We focused on the enforcement of policies that are provided by the OSNs in a TPA-independent manner. However, our approach is also able to enforce compliance with policies provided by legislators, TPAs, or end users. In terms of the latter, a solution might be to retrieve the users' privacy settings from the OSN, translate those into enforceable ECA rules, and enforce compliance with them.

While this work focused on the preventive enforcement of policies, enforcement might also be detective. This can be easily achieved by deploying policies that explicitly allow for policy violations (ECA-Action: Allow unmodified execution), but trigger some form of logging at the same time.

Seal of Compliance. As of now, our solution does not indicate to users whether some TPA is running on a trusted PaaS. We envision the presence of a "Seal of Compliance" (similar to SSL indicators) that indicates to users whether or not a particular TPA runs on a trusted PaaS that enforces compliance. Such an indicator could be built into the web browser, e.g. in the form of a browser plugin. Alternatively, a "Seal of Compliance" could be displayed to users on the TPA's site within the OSN, or on the OAuth authentication site that is displayed to users upon initial login to the TPA.

VI. Related Work

xBook [23] is most similar to our approach. It provides a framework for developing privacy-preserving TPAs. Thereby, TPAs are split into client-side and server-side components. All communication between them as well as to any external entities is mediated through xBook. Both information flow tracking and policy enforcement is performed at the JavaScript layer. Similar to our approach, TPAs must be executed on a trusted platform which provides the xBook framework. However, xBook demands TPAs to be rewritten from scratch, adopting the notion of components and communicating solely using the xBook library framework.

MUTT [20] is a set of PaaS extensions that ensure that the users' access control rules, as specified within the OSN, are also enforced within TPAs. For instance, MUTT ensures that a TPA only shares a user's picture with legitimate friends in correspondence with the user's access control settings within the OSN. For this, MUTT dynamically tracks and controls the flow of data within the TPA. Our approach is more expressive by allowing to enforce compliance with arbitrary policies that go beyond access control settings provided by the OSN.

Several works address the problem of untrustworthy TPAs by limiting their access to sensitive user data in the first place [1, 3, 5, 21, 22]: Anthonysamy et al. [1] implement a trusted proxy through which all requests to the Facebook API are redirected. Responses from Facebook are only forwarded to the TPA after sensitive data was sanitized in correspondence with the user's configuration. PoX [5] takes a similar approach but implements the proxy within the user's browser, thus avoiding an additional trusted third party. A fallback solution exists for browsers and TPAs that do not support PoX. Cheng et al. [3] propose to run parts of the TPA within the domain of the OSN rather than on an external server, thus limiting the amount of user data being released to the outside. The approach by Shehab et al. [21, 22] allows users to define for each TPA and data type whether access is allowed. All of these approaches might compromise the utility of TPAs, since essential data to provide proper service might be missing. Our approach only compromises utility in case of policy violations. Also, it goes a step further by controlling how the TPA might handle user data after it was received.

VII. Conclusions

Users of Online Social Networks (OSNs) increasingly also use third-party applications (TPAs), which provide their social experience on the basis of sensitive user data accessed through the OSN. In this work we provide mechanisms to ensure the compliance of such TPAs with security and privacy policies in a generic and transparent manner. Using our approach, TPAs are kept from misusing sensitive user data, e.g. by releasing it to advertisement services or by not keeping it up-to-date.

We achieve this compliance of TPAs by building usage control technology into PaaS platforms on which the TPAs are assumed to be executed. Whenever requests for sensitive user data at the OSN are performed, our infrastructure deploys corresponding security and privacy policies. These policies are expected to be provided in a machine-readable format by the OSN operator. Once sensitive data was received by the TPA, our infrastructure ensures compliance with aforementioned policies. By leveraging information flow control technology, our approach even enforces policies correctly in case the sensitive data is copied to different storage locations within the TPA. Notably, our approach is transparent for TPAs: no changes to existing applications are required.

We implement our approach for Facebook and the Java Runtime Environment, thus allowing to ensure the compliance of TPAs with Facebook's Platform Policy. Concretely, we develop and deploy a sample BirthdayCalendar application that displays a user's friends' birthdays. We show how the policy "you may cache the content for up to 24 hours" maps to a machine-readable format and how it is actually enforced. On the basis of this use case, we provide a preliminary performance evaluation of our prototype, revealing an overall runtime performance overhead between 15% and 41%.

As future work, we plan to improve the runtime performance of our approach (cf. § IV). We further seek to instantiate our approach for other SEE, such as PHP. The goal is to cover

a broader set of TPAs, and hence, to be able to evaluate our approach on a more comprehensive set of applications as well as security and privacy policies.

Acknowlegdements. This work was sponsored by the German Federal Ministry of Education and Research (grant 01|S12057).

References

[1] P. Anthonysamy, A. Rashid, J. Walkerdine, P. Greenwood, and G. Larkou. "Collaborative Privacy Management for Third-party Applications in Online Social Networks". In: *Proc. 1st Workshop on Privacy and Security in Online Social Media*. ACM, 2012, 5:1–5:4.

[2] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. "Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders". In: *33rd Int'l Conf. Software Engineering*. ACM, 2011.

[3] Y. Cheng, J. Park, and R. Sandhu. "Preserving User Privacy from Third-party Applications in Online Social Networks". In: *22nd Int'l Conf. World Wide Web*. 2013.

[4] D. Hardt (Ed.) *RFC6749: The OAuth 2.0 Authorization Framework*. Oct. 2012.

[5] M. Egele, A. Moser, C. Kruegel, and E. Kirda. "PoX: Protecting Users from Malicious Facebook Applications". In: *Computer Communications* 35.12 (2012).

[6] Facebook, Inc. *Facebook Platform Policy*. Accessed: 2016/02/02. Mar. 2015. URL: http : / / developers . facebook.com/policy.

[7] A. Fromm, F. Kelbert, and A. Pretschner. "Data Protection in a Cloud-Enabled Smart Grid". In: *Smart Grid Security*. Vol. 7823. LNCS. Springer, 2013, pp. 96–107.

[8] H. Gao, J. Hu, T. Huang, J. Wang, and Y. Chen. "Security Issues in Online Social Networks". In: *IEEE Internet Computing* 15.4 (July 2011), pp. 56–63.

[9] J. Graf, M. Hecker, and M. Mohr. "Using JOANA for Information Flow Control in Java Programs - A Practical Guide". In: *Proc. 6th ATPS*. 2013.

[10] M. Harvan and A. Pretschner. "State-Based Usage Control Enforcement with Data Flow Tracking using System Call Interposition". In: *3rd International Conf. on Network and System Security*. Oct. 2009, pp. 373–380.

[11] M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter. "A Policy Language for Distributed Usage Control". In: *Computer Security – ESORICS 2007*. Vol. 4734. LNCS. Springer, 2007, pp. 531–546.

[12] F. Kelbert and A. Pretschner. "A Fully Decentralized Data Usage Control Enforcement Infrastructure". In: *Applied Cryptography and Network Security*. Vol. 9092. LNCS. Springer, 2015, pp. 409–430.

[13] F. Kelbert and A. Pretschner. "Data Usage Control Enforcement in Distributed Systems". In: *3rd Conf. on Data and App. Security and Privacy*. ACM, 2013.

[14] F. Kelbert and A. Pretschner. "Decentralized Distributed Data Usage Control". In: *Cryptology and Network Security*. Vol. 8813. LNCS. Springer, 2014, pp. 353–369.

[15] F. Kelbert, F. Shirazi, H. Simo, T. Wüchner, J. Buchmann, A. Pretschner, and M. Waidner. "State of Online Privacy: A Technical Perspective". In: *Internet Privacy*. acatech Studie. Springer, 2012, pp. 189–279.

[16] P. Kumari and A. Pretschner. "Model-Based Usage Control Policy Derivation". In: *Engineering Secure Software and Systems*. Vol. 7781. LNCS. Springer, 2013.

[17] LinkedIn Corp. *API Terms of Use*. Accessed: 2016/02/02. Feb. 2015. URL: http : / / developer . linkedin.com/documents/linkedin-apis-terms-use.

[18] E. Lovat, A. Fromm, M. Mohr, and A. Pretschner. "SHRIFT: System-Wide HybRid Information Flow Tracking". In: *ICT Systems Security and Privacy Protection*. Vol. 455. IFIP Advances in Information and Communication Tech. Springer, 2015, pp. 371–385.

[19] A. Pretschner, E. Lovat, and M. Büchler. "Representation-Independent Data Usage Control". In: *Data Privacy Management and Autonomous Spontaneus Security*. Vol. 7122. LNCS. Springer, 2012.

[20] A. Shakimov and L. P. Cox. "MUTT: A Watchdog for OSN Applications". In: *1st Conference on Timely Results in Operating Systems*. ACM, 2013, 6:1–6:14.

[21] M. Shehab, A. Squicciarini, G.-J. Ahn, and I. Kokkinou. "Access Control for Online Social Networks Third Party Applications". In: *Computers & Security* 31.8 (2012).

[22] M. Shehab, A. Squicciarini, and G.-J. Ahn. "Beyond User-to-User Access Control for Online Social Networks". In: *Information and Communications Security*. Vol. 5308. LNCS. Springer, 2008, pp. 174–189.

[23] K. Singh, S. Bhola, and W. Lee. "xBook: Redesigning Privacy Control in Social Networking Platforms". In: *18th USENIX Security Symposium*. 2009, pp. 249–266.

[24] Tumblr, Inc. *Application Developer and API License Agreement*. Accessed: 2016/02/02. Jan. 2014. URL: http://www.tumblr.com/docs/en/api_agreement.

[25] T. Wüchner and A. Pretschner. "Data Loss Prevention Based on Data-Driven Usage Control". In: *IEEE 23rd Int'l Symp. on Software Reliability Engineering*. 2012.