# Enforcing Expressive Accountability Policies

Ronan-Alexandre Cherrueau, Mario Südholt

Ascola team; Mines Nantes, Inria, LINA
Département Informatique, École des Mines de Nantes, Nantes, France
Email: firstname.lastname@mines-nantes.fr

***Abstract*—Accountability policies for the enforcement of the responsible stewardship of personal data have to support the gathering of information at all levels of the service stack and across different policy domains, for instance, for the retrospective enforcement of transparency and remediation properties. Existing approaches to accountability, however, often do not meet these requirements and corresponding implementation support is lacking.**

**In this paper we show how expressive accountability policies can be defined in terms of policy domains, accessible data at all levels of the service stack, and preventive and retrospective mechanisms. Additionally, we present a notion of accountability schemes that support the constructive implementation of our accountability policies. Finally, we motivate and apply our approach in the context of real-world attacks to OAuth-based authorization and authentication protocols.**

## I. Introduction

One of the main objectives of accountability in software systems consists in the responsible stewardship of personal data [1], [2]. Ensuring accountability (*e.g.,* obligations of transparency in data usage, attributability of responsibility, remediation capabilities in the case of abuse) in large-scale multiparty application, such as web applications and service-based applications in the cloud, is notoriously difficult. Furthermore, in contrast to the rich body of work on security policies few general approaches to accountability policies exists and no general constructive support for their definition and implementation is available.

In order to illustrate the challenges of accountability enforcement, consider the multiparty web application shown in Fig. 1. This application consists of three services (the boxes). A cloud storage service [3] acts as a resource server and shares user's data. A third-party application enables users to access the cloud storage. Finally, an OAuth 2.0 authorization server [4] allows user to grant third-party applications access to their storage using a token, without sharing their login credentials.

In this context, the involved parties may be interested in enforcing the following basic accountability requirements:

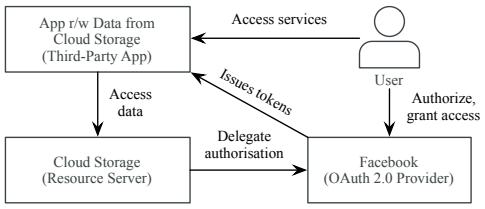


Fig. 1. The OAuth 2.0 Big Picture

- Users require that their data is correctly secured during exchanges between involved parties. Consequently, no unauthorized parties should gain access to her data.
- The OAuth provider requires that both the user and third-party application are correctly identified. Thus, it grants third-party correct access to the cloud storage.
- The third-party application requires that user presents the correct token. Therefore, it ensures that it only performs actions authorized by the user.

The challenge consists in satisfying these requirements in the context of heterogeneous services hosted in the cloud, where data may be leaked easily. For instance, an access token can be easily obtained by exploiting a not correctly secured or malicious third-party application. Furthermore, these attacks have subtle properties that make their identification and remediation very difficult:

- Third-party applications have to secure OAuth information on the implementation level according to the OAuth standard, so information cannot be accessed through the service interface.
- The two attacks can be combined in scenarios that involve different authorization processes and cannot be identified and handled correctly solely based on an individual process.

The first property calls for multi-level accountability policies that are defined in terms of service implementations in addition to service interceptors and orchestrations. The second requires accountability policies to be defined over different policy domains. However, current accountability frameworks do not support the former and provide only limited support for the latter.

In this paper, we provide support for both properties through a notion of multi-level accountability policies defined over a notion of scope that spans accountability domains. Concretely, we provide the following contributions:

1) We motivate the need of such of policies in the context of advanced attack scenarios involving OAuth-based authorization and authentication, services for which it is used frequently by all major web and cloud players (Sec. II).
2) We present a language for the definition of accountability policies over horizontal service compositions and multi-level service stacks (Sec. III).
3) Based on this language, we propose a notion of accountability schemes that act as constructors of general accountability policies. Since schemes are parameterized, specialized schemes may directly reuse functionality from more general ones. Furthermore, we introduce two sets of schemes for transparency and remediation purposes,

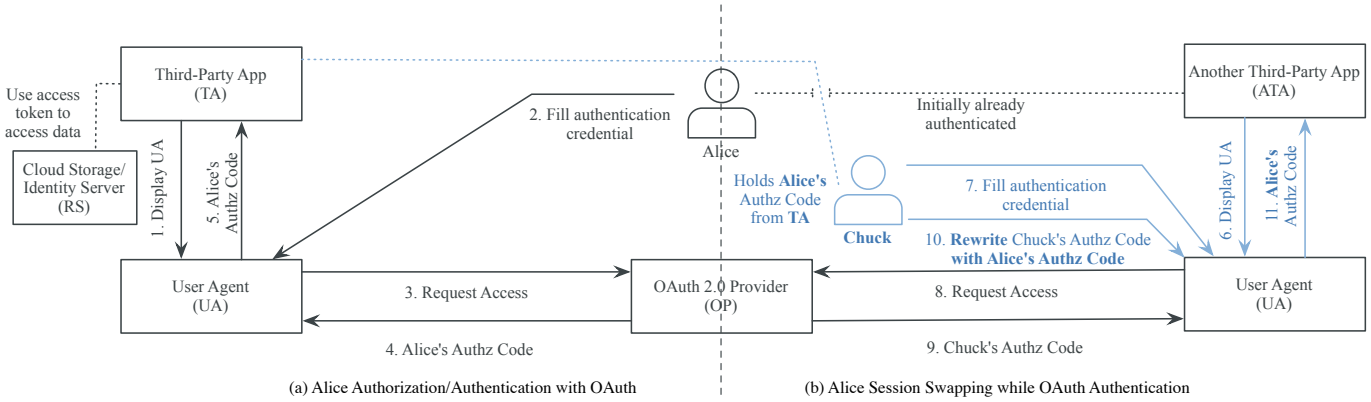


Fig. 2. OAuth, Principle (a) and Attack (b)

two of the main accountability properties. We validate these accountability constructors by applying them to the construction of accountability policies that handle the motivating OAuth attack scenarios (Sec. IV).

4) We compare our approach to related work in Sec. V and conclude in Sec. VI.

## II. Motivation: Accountability Policies for OAuth-based Authorization

We now consider in more detail the enforcement of accountability policies in the context of OAuth authorization and authentication protocols (*i.e.,* single sign-on [5]). We first briefly introduce the reader to the OAuth framework and several real-world attack scenarios that exploit security vulnerabilities of OAuth-based implementations. Based on these scenarios we then motivate that accountability properties, in particular, transparency and remediation properties, require means for the definition of cross-domain relationships and access to the different levels of the service stack (implementation, interceptors and service interfaces/orchestration).

### A. The OAuth Framework

OAuth [4] is an authorization framework that allows users to grant third-party application access to their resources without sharing their login credentials. The user authenticates to an OAuth provider, which then delivers access tokens to authorized third-party applications. The third-party application uses this token to access to resources on behalf of the user.

*1) Authorization code grant flow:* In order to third-party applications obtain an access token, the OAuth standard offers four grant flows. The most commonly used one is the authorization code grant flow, henceforth simply called grant flow and illustrated in Figure 2(a). This flow is started by a third-party application (TA) directing the user to the User Agent (UA) of the OAuth provider (OP). The UA is typically a web-browser page with a login form. The user then fills the authentication form with her credentials and decides whether or not to grant her authorization to TA (step 2). As a result, the request made by the UA includes the user's credentials and decision. If the user agreed to TA's request, the OP generates an authorization code and sends it back to the UA (steps 3, 4). Finally, the UA redirects the authorization code to the TA using a specific URL (step 5). At the end, the authorization code is redeemable for an access token. The TA can thus access the user's personal resources without ever seeing the user's credentials.

*2) Single sign-on:* The grant flow is also commonly used for authentication as a form of a web-based single sign-on (SSO) protocol [5]. In this case, the resource to be accessed is the user's personal information that permits her identification. The TA issues a token for her identity and associates this identity to her account in its internal user account data. It then employs the user's identity to look up its internal user account data and authenticates her.

### B. Attack scenarios and accountability

We motivate accountability challenges in the context of a class of attacks, so-called session swapping attacks, on OAuth-based implementations. Such attacks have targeted social networking sites using single sign-on (SSO) protocols over the last years. These attacks allow the malicious reuse of OAuth grants of legitimate users [5], [6], [7]. There are two ways to perform session swapping: either by exploiting the absence of session-identifying state information, or using a malicious third-party application.

*1) State-based SSO Session Swapping:* In order to abuse the identity association in SSO, an attacker (Chuck) forges an authorization grant request and tricks Alice into filling in her credentials [5], [6]. As a result, TA associates Chuck's identity with Alice's account in its internal user account data. Chuck can then connect to TA under Alice's identity.

*Accountability challenges:* In this exploit, Chuck swaps sessions and fools the third-party application by pretending to be Alice. Chuck's exploit, therefore, violates accountability with respect to Alice's personal data. To avoid this exploit, a unique state for each new grant flow can be employed. This state has to be saved securely within TA's implementation and then be carried along with other parameters during the rest of the grant flow, thus identifying who has initiated the grant flow. At the last step of the flow (Figure 2(a), step 5), the third-party application turns down the authorization for Alice's identity if Alice is not the initiator.

In this case, enforcing Alice's accountability requirements, such as the preservation of her credentials and the data stored by a resource provider, requires knowledge about the presence of state parameters in the grant flow (*i.e.,* at the orchestration level from a service point of view), and whether the state is saved securely in the third-party application (which is only available at the service implementation level). This is difficult because third-party applications and OAuth providers are not *transparent* and do not provide corresponding information through their interfaces. In addition, accountability policy enforcement requires *remediation* if personal data is leaked because of a violation. This can be done in two different ways. First, in a preventive manner, by dynamically modifying the multi-party application and introducing the state parameter at the implementation and orchestration levels. Second, in a retrospective manner, by revoking access tokens and notifying all parties that a session swap happened. As we show in Sec. IV, our framework supports both ways to enforce accountability.

*2) Session Swapping based on compromised apps:* Even if the state is introduced in the grant flow, Chuck can steal Alice's identity with the complicity of a malicious third-party application [6]. For the rest of this example, consider that the TA in Figure 2(a) is a malicious third-party application and ATA in Figure 2(b) is the third-party application on which Chuck will perform the attack. Alice is authenticated on both TA and ATA using different instances of a SSO protocol involving the same OAuth provider. Chuck holds Alice's authorization code from the malicious app (TA) and triggers a grant flow (step 6) starting at ATA. Chuck manipulates the authorization response and substitutes his access code for Alice's one (steps 10, 11). This code is then exchanged by ATA for a token and ATA uses this token to identify the user in its internal user account data. But since the token binds Alice's identity, Chuck is now connected to ATA under Alice's identity.

*Accountability challenges:* Just like the previous example, Chuck fools the third-party application by pretending to be Alice and thus violates the accountability policy on the usage of Alice's personal data. To prevent this attack all third-party applications must provide their application ids along with every request to exchange an authorization code for an access token. Moreover, the OAuth provider must validate whether an authorization code has been issued to the particular third-party applications. In addition, best practices require that the third-party application shall be authenticated beforehand.

This attack, while technically different from the previous one, violates the same accountability properties. Note that both attacks share the same context, *i.e.,* session swapping in an authorization code grant flow. The common context can be seen as a *scope* during which the accountability policy has to be enforced and this context is independent from transparency and remediation mechanisms. Rather than defining the targets of violations and how to remedy them, the scope defines the context where information has to be gathered.

In summary, these two attacks provide strong evidence that accountability policies in multi-party applications benefit from the following support:

- *Transparency* mechanisms that identify the elements necessary for the violation of the policy and where the violation appears. This identification could be done at runtime and requires access to information at the three levels of service definitions (implementation, interface and orchestration).
- *Remediation* mechanisms that enable violations to be corrected. Remediation also requires the modification of functionalities at the three service levels and should be applicable in preventive or retrospective manner.
- *Scope* definitions to describe the contexts where information has to be gathered, where violations happen, and where remediation actions have to be applied.

// Services, interceptors, methods

$S \quad := \quad \ldots \mid \mathrm{i}(\overline{E}) \mid \ldots \mid \mathrm{m}(\overline{E}) \mid \ldots \mid \mathrm{s}(\overline{E}) \mid \ldots$

// Composition levels: services, interceptors, implem.

$L \quad := \quad \mathtt{s} \mid \mathtt{c} \mid \mathtt{m}$

// Level-aware compositions and patterns

$P \quad := \quad S \mid P;P \mid P \to P \mid P|P \mid P* \mid (\ P\ ) \mid \_$
$\mid \quad P\&P \mid P_L \mid P@AI \mid Pred$

// $E$ argument expr.; $AI$ action id.

Fig. 3. Services, compositions and patterns: the service DSL

$Pred \ := \ \textbf{\textit{exists}}(E,E) \mid \textbf{\textit{isResponsible}}(S) \mid \textbf{\textit{dataFlow}}(S)$
$\mid \ !Pred \mid Pred \wedge Pred \mid Pred \vee Pred$

// $E$, $S$: cf. Fig. 3

Fig. 4. Accountability and security predicates

## III. Accountability Policy Language

We propose a language for the construction of accountability policies that consists of three components:

- Service compositions and patterns over such compositions
- Accountability-specific predicates
- Generic schemes that provide scopes and enforcement actions for policies

Service compositions are defined as shown in Fig. 3. The elementary constituents (non-terminal $S$) of service compositions are chosen from three levels within the service stack: invocations of services, interceptors (that are part of most real-world service-oriented infrastructures) and implementation methods. Composition expressions may be restricted to these three levels ($L$): $\mathtt{s}$:services, $\mathtt{c}$:interceptors, $\mathtt{m}$:implementation. Non-terminal $P$ defines service compositions and patterns (that can be matched at runtime or analyzed statically): the expressions in the first line enable the definition of regular compositions over elementary services and the wildcard '_' that stands for arbitrary elementary constituents. These are standard except for the two sequentialization operations $P_1; P_2$ and $P_1 \to P_2$: the former expresses that $P_2$ follows $P_1$, possibly encompassing interleaved parts, while the latter requires that $P_2$ immediately follows $P_1$: $x_s \to y_m$ defines, for example, that $y$ is the method implementing the service $x$ (because it refers to the first method called after the call to $x$). The second and third line define pattern-specific expressions. These include pattern intersections $P\&P$, the restriction of patterns to certain levels $P_L$, the application of actions once a pattern is matched $P@AI$; finally, (accountability) predicates $Pred$ can be used in pattern intersections.

// Policy scopes
$PS$ := ***pscope*** $SI$ < $SPAR$ > $P$

$SPAR$ := $\epsilon$ | $P$

// Policy elements

$PE$ := ***pelem*** $SI$ < $EPAR$ > { $\overline{PS}$ $\overline{AC}$ } $IN$

$IN$ := ***inst*** $PE$ | $\epsilon$

$EPAR$ := $\epsilon$ | $\overline{PS}$ | $\overline{AI}$

// Actions

$AC$ := ***action*** $AM$ $AI$ { $AB$ }

$AM$ := `after` | `around` | `before`

$AB$ := $\epsilon$ | `proceed` | $S_L$ | $\overline{AB}$

// $SI$ scope/scheme id., $PI$ pattern id
// $L$, $P$, $S$: cf. Fig. 3

Fig. 5. Policy language

Accountability predicates (Fig. 4) enable propositional formulas to be defined over elementary predicates that allows one to test for, for instance, the existence of a data element, such as a value, in another one, for instance an argument list (***exists***), the service being responsible for initiating a computation (***isResponsible***) or whether some data satisfy a data-flow relationship from one service to another (***dataFlow***). Patterns involving accountability predicates can be analyzed statically, *e.g.,* for predicates that are satisfied at specific points during the execution represented by the patterns. This feature is, however, beyond the presentation in this paper.

We propose two main abstractions from which accountability policies can be built (Fig. 5): policy scopes and policy schemes. Scopes, introduced by ***pscope***, essentially encapsulate a (service composition) pattern that defines a (potentially non-contiguous) period of time during which a policy should be applied. A scope is named and may be parameterized with pattern elements. Scopes may be used, in particular, to express policies that span different policy domains. A policy scheme, introduced by ***pscheme***, applies actions $AC$ within a policy scope: to this end action ids $AI$ are called from within the patterns of scopes as defined by the $P@AI$ pattern constructor. Policy schemes may inherit definitions from other schemes by specifying an (optional) instantiation clause ($IN$): schemes may thus be used to build hierarchies of more general and more specific policies.

This language is based on a former more general but lower-level language for the transformation of service compositions [8]. Compared to that language, our policy language features a simplified but more regular service DSL, and introduces three new concepts: a predicate language for accountability, as well as scopes and schemes for the construction of executable accountable policies. The accountability policy language can be implemented on top of our previous service transformation language. An implementation of the latter, called SAdapt, that allows the transformation of services implemented using Apache's CXF infrastructure is available from [9].

## IV. Accountability Policy Schemes

In this section, we present a new notion of accountability schemes based on the policy language introduced in the previous section. Accountability schemes provide general support for the enforcement of accountability properties in terms of scopes that may span different policy domains and accountability actions. General parameterized schemes may be specialized by parameter instantiation. Furthermore, we introduce two sets of accountability schemes, cf. Fig 6, one for transparency-related policies, another one for remediation policies. Finally, we show how to harness instances of these general schemes in order to enforce accountability policies in the context of the OAuth-based attacks introduced in Sec. II.

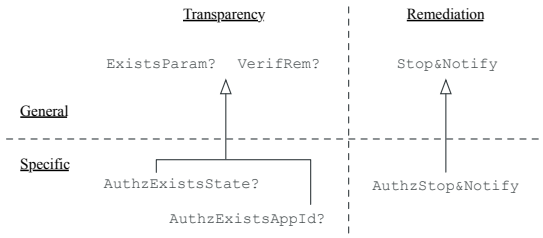


Fig. 6. Concrete Schemes for Accountability Policies Enforcement

### A. General Schemes for Accountability Policy Enforcement

Figure 6 lists three schemes that provide general common identification and correction mechanisms for the enforcement of accountability policies. The first one, named `ExistsParam?` identifies if a mandatory parameter is present or not. Its scope and action that enables preventive and retrospective remediation are undefined, they have to be provided by instantiations. The second transparency scheme, named `VerifRem?` is another transparency scheme that verifies if a remediation has been applied correctly by querying system properties. The last general scheme, named `Stop&Notify` is a retrospective remediation scheme. It retrospectively corrects a policy violation by stopping the current process and notifying all participants that a policy violation has happened.

Fig. 7 defines the `ExistsParam?` transparency scheme. The scheme takes five parameters. `X` and `Y` are query delimiters that define where the violation appears, possibly within different policy domains. They also permit developers to specify exactly where to search for the mandatory parameter. For instance, the parameter could occur at the implementation level as part of a method call, or at the orchestration level in a service call. The scheme in Figure 11, which instantiates the `ExistsParam?`, shows how to specialize the two parameters to find the third-party application identifier in the context of the OAuth three-party authentication. In this example, the scheme searches for the existence of an application identifier at the orchestration level (as defined by the 's' index). In addition to `X` and `Y` parameters, `param` is used with the ***exists*** predicate to identify the existence of the mandatory parameter. So, in the scope delimited by `X` and `Y`, if the `param` value is present, the system triggers the `Exists` action, or otherwise, the `NotExists` action. There is no assumption on when the actions are triggered (*i.e.,* `before`, `around` or `after`); at the time of policy instantiation, the developer could choose the best strategy for her action. For instance, the scheme in Figure 11 specifies to apply remediation preventively: it provides notice that the application identifier exists by triggering a

before action. To the contrary, Fig. 10 which is designed to indicate that the secure state does not exist, triggers its action with after for retrospective remediation.

```
pscheme ExistsParam? <
    X,Y,                // Query Starter, Finisher
    param,              // Mandatory Param
    Exists, NotExists, // Actions
    > {
  // Identify Param Exists:
  pscope CheckParamExists <  >
    [X & exists(param, args)]@Exists ; _{s,c,i} ; Y
  // Identify Param Not Exists:
  pscope CheckParamNotExists <  >
    [X & !exists(param, args)]@NotExists ; _{s,c,i} ; Y
}
```

Fig. 7. ExistsParam? General Transparency Scheme

Figures 8 defines the general schemes VerifRem? and Stop&Notify. As in the case of ExistsParam?, the parameters of those two schemes enable covering a large number of enforcement policies for accountability policies. Thus, the Query parameter of VerifRem? permits the verification whether remediation has taken place by simply querying the system about the effect of the remediation. Analogously, the general remediation of Stop&Notify is a common remediation in accountable systems. The remediation uses a Query parameter to know when and where to stop the current process.

```
pscheme VerifRem? < Query, IsVerified > {
  pscope CheckIsVerified <  > Query@IsVerified
}
pscheme Stop&Notify < Query, AfterNotification > {
  pscope CheckToStop <  >
    Query@Stop&NotifyParties@AfterNotification
  action around Stop&NotifyParties {
    notify("Violation of ", Query); System.exit(0); }
}
```

Fig. 8. VerifRem? General Transparency Scheme and Stop&Notify General Remediation Scheme

As can be seen, our language permits to abstract mechanisms for the enforcement of accountability policies. It enables the abstract definition of transparency mechanisms and thus denotes a whole range of situations in which policy can be applied. It also allows the abstract definition of remediation mechanisms. Hence, the language offers tools to easily correct policy violations in a preventive or retrospective mode. Our approach is not limited in any way to schemes pertaining to transparency and remediation. For instance, Fig. 11 provides an example of a scheme that gathers information and attributes responsibility for a violation, another major accountability property. Overall, we aim (as future work) for a complete library for the enforcement of accountability policies.

### *B. Specific Schemes for OAuth-based policy enforcement*

In the previous section we have presented general accountability schemes. We now specialize those schemes for the identification of accountability-relevant contexts and the remediation of accountability violations in the context of OAuth-based third-party authentication.

For abstraction reason, schemes of the previous section are defined without providing scopes. Thus, a scheme like ExistsParam? is applicable independently of a context. However, we now have to apply schemes in the context of OAuth-based third-party authentication. to this end Fig. 9 defines the scope of an authorization code grant flow in OAuth. This scope is then used by two specialized schemes, AuthzExistsState? (Fig. 10) and AuthzExistsAppId? (Fig. 11), that respectively return whether a state parameter and an application identifier parameter exists.

```
pscope AuthzFlow < X >
  displayUA(args, K)_s ; X ; K(code, args')_s
```

Fig. 9. Scopes for OAuth multi-party authentication

```
pscheme AuthzExistsState? < > inst ExistParam? <
    reqAcc(args, K')_s, K'(code, args')_s
    "state",
    LogState, LogNoState >

  // Limits the scope:
  pscope AuthzCodeFlow < CheckParamExists >
  pscope AuthzCodeFlow < CheckParamNotExists >

  // Defines actions:
  action after LogState {
    log("In AuthFlow") ; log("State", args) }
  action after LogNoState {
    log("In AuthFlow") ; log("No State", args) }
}
```

Fig. 10. AuthzExistsState? Specific Transparency Scheme

It is important to notice that Fig. 11 introduces another scope named Register which identifies if we are not only in the context of authorization code grant flow, but also ahead of third-party application registration. This lets the AuthzExistsAppId? scheme know if the fault of the absence of the application identifier is imputable to OAuth provider (in case of non registration) or to the third-party application.

## V. Related work

We now consider three classes of related work: (i) general approaches for the definition of accountability policies, (ii) methods for specific accountability properties, and (iii) approaches limited to security and privacy policies.

There is only a small body of work of general approaches to the definition of accountability policies. Benghrabrit *et al.* [10], for example, propose a high-level temporal-logic based language for accountability obligations. A lower-level language that is based on PrimeLife Policy Language (PPL) [11] is used for their enforcement. With their language, a developer can easily elaborate high accountability obligation in an elegant manner. But, using their approach cross-domain properties can only be encoded but not be expressed directly. No support for multi-level properties is provided. While their approach can be used for property verification it does not include implementation support. Many other approaches target formal models for accountability without any implementation support [12], [13],

```
pscope Register < >
  OP.register(TA_name, TA_K)_s ;
    OP.addAppId(TA_name ↦ id)_im ; TA_K(id)_s

pscheme AuthzExistsAppId? < > inst ExistParam? <
    reqAcc(args, K')_s, K'(code, args')_s
    "client_id",
    LogAppId, LogNoAppId >

  // Limits the scope with possible registration:
  pscope (Register@LogRegistration | _s,c,i) ;
    AuthzCodeFlow < CheckParamExists >
  pscope (Register@LogRegistration | _s,c,i) ;
    AuthzCodeFlow < CheckParamNotExists >

  // Defines actions:
  action before LogRegistration {
    log("App Register", TA_name, id) }
  action before LogAppId {
    log("In AuthFlow") ; log("AppId", args) }
  action before LogNoAppId {
    log("In AuthFlow") ; log("No AppId", args) }
}
```

Fig. 11. `AuthzExistsAppId?`: Specific Transparency and Attributability Scheme

[14]; furthermore, none of these address our concerns of cross-domain, multi-level accountability properties.

There is quite a large number of articles that present support for specific accountability properties and thus do not share our objective of generality. However, some of these approaches support, albeit only implicitly, cross-domain and multi-level accountability properties. Sundareswaran *et al.* [15], for instance, presents a model and an implementation method for secure logging in distributed environments that can be used to log service interactions at the orchestration level but also at the implementation level. However, the logging has to be programmed and configured explicitly at all levels. The approach is therefore much less declarative than ours.

Finally, note that more traditional approaches to the definition and implementation of security and privacy policies, *e.g.,* [16], [17], do not address our concerns in that they do not cover many aspects of accountability, notably remediation, and only harness preventive means for property enforcement.

In summary, the key objectives of our approach — preventive and retrospective support for cross-domain and multi-level accountability policies, as well as constructive implementation support for different types of accountability properties — are not met by previous work.

## VI. Conclusion

We have motivated that accountability policies require access to all levels of service-based implementations of applications, should include explicit means for the definition of cross-domain policies and provide constructive means for the implementation of a wide variety of of accountability properties, features that are missing in existing approaches.

We have provided an approach that addresses these objectives explicitly through a language for the definition of expressive regular policies over accountability predicates applicable at all levels of the service stack. Furthermore, we have presented hierarchies of constructive schemes for the implementation of policies for transparency and remediation properties that are implemented in terms of our accountability policy language. Finally, we have shown how to harness the accountability schemes to tackle real-world violations of accountability properties arising from security vulnerabilities of OAuth-based authorization and authentication protocols.

The main tracks for future work include (i) the specialization of our existing implementation for more general service manipulation schemas to the accountability schemes introduced here and (ii) the development of analysis techniques of accountability properties defined using our policy language.

## VII. Acknowledgment

This work was partially supported by the Cloud Accountability project (A4Cloud, EC grant no. 317550, `www.a4cloud.eu`).

## References

[1] S. Pearson, "Toward accountability in the cloud," *IEEE Internet Computing*, vol. 15, no. 4, 2011.

[2] J. Feigenbaum, A. Jaggard, R. Wright, and H. Xiao, "Systematizing 'Accountability' in Computer Science," YALEU/DCS/TR-1452, Yale University, New Haven CT, Tech. Rep., 2012, Version of Feb. 17, 2012.

[3] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran, "Object storage: The future building block for storage systems," IEEE, pp. 119–123, 2005.

[4] D. Hardt, "The OAuth 2.0 Authorization Framework," RFC 6749 (Proposed Standard), Internet Engineering Task Force, Oct. 2012.

[5] S.-T. Sun and K. Beznosov, "The devil is in the (implementation) details: an empirical analysis of oauth sso systems," 2012.

[6] T. Lodderstedt, M. McGloin, and P. Hunt, "OAuth 2.0 threat model and security considerations," RFC 6819 (Informational), Internet Engineering Task Force, 2013.

[7] C. Bansal, K. Bhargavan, and S. Maffeis, "Discovering concrete attacks on website authorization by formal analysis," pp. 247–262, 2012.

[8] R.-A. Cherrueau, M. Südholt, and O. Chebaro, "Adapting workflows using generic schemas," p. 6, Dec. 2013.

[9] The Ascola team, "Advanced Service Composition," http://a4cloud.gforge.inria.fr/doku.php?id=start:advservcomp.

[10] W. Benghabrit, H. Grall, J.-C. Royer, M. Sellami, M. Azraoui, K. Elkhiyaoui, M. Önen, A. Santana De Oliveira, and K. Bernsmed, "A Cloud Accountability Policy Representation Framework," in *CLOSER*, 2014.

[11] S. Trabelsi, J. Sendor, and S. Reinicke, "Ppl: Primelife privacy policy engine," in *POLICY*, 2011.

[12] S. Etalle and W. H. Winsborough, "A posteriori compliance control," in *Proc. of the 12th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '07. ACM, 2007.

[13] R. Jagadeesan, A. Jeffrey, C. Pitcher, and J. Riely, "Towards a theory of accountability and audit," in *Proc. of the 14th European Conf. on Research in Computer Security*, ser. ESORICS'09. Springer, 2009, pp. 152–167.

[14] R. K üsters, T. Truderung, and A. Vogt, "Accountability: Definition and relationship to verifiability," in *Proc. of the 17th ACM Conf. on Computer and Communications Security*, ser. CCS'10. ACM, 2010.

[15] S. Sundareswaran, A. C. Squicciarini, and D. Lin, "Ensuring distributed accountability for data sharing in the cloud," *Dependable and Secure Computing*, 2012.

[16] D. Le Métayer, "A formal privacy management framework," in *Formal Aspects in Security and Trust*, ser. LNCS. Springer Berlin Heidelberg, 2009, vol. 5491.

[17] M. Y. Becker, A. Malkis, and L. Bussard, "S4p: A generic language for specifying privacy preferences and policies," Microsoft Research, Tech. Rep. MSR-TR-2010-32, 2010.