A Trustworthy Usage Control Enforcement Framework

Ricardo Neisse
Fraunhofer IESE, Germany
ricardo.neisse@iese.fraunhofer.de

Alexander Pretschner
Karlsruhe Institute of Technology, Germany
pretschner@kit.edu

Valentina Di Giacomo
Engineering, Italy
valentina.digiacomo@eng.it

***Abstract*—Usage control policies specify restrictions on the handling of data after access has been granted. We present the design and implementation of a framework for enforcing usage control requirements and demonstrate its genericity by instantiating it to two different levels of abstraction, those of the operating system and an enterprise service bus. This framework consists of a policy language, an automatic conversion of policies into enforcement mechanisms, and technology implemented on the grounds of trusted computing technology that makes it possible to detect tampering with the infrastructure. We show how this framework can, among other things, be used to enforce separation-of-duty policies. We provide a performance analysis.**

***Keywords*-Data usage control**

I. Introduction

Usage control requirements stipulate the handling of data after access is granted and generalize access control requirements. Typical requirements include "delete data within thirty days" and "actions one and two on a specific item must not be performed by the same individual". The handling of data is reflected by events at different levels of abstraction including the operating system (OS; events: system calls), enterprise service bus (ESB; events: messages), windowing system (X11; events: network packets), and so on.

In this paper we introduce a usage control policy enforcement framework that was developed as part of an EU-funded research project [1]. In this project, our framework was used to demonstrate two application scenarios in a hospital and an insurance company. Requirements in these scenarios include Separation of Duty (SoD) and privacy protection through identity anonymization. Due to space restrictions, we concentrate on an SoD example in this paper.

The more general setting is as follows. Business processes are formalized as sequences of events. Control objectives (e.g SoD: "more than one person should be involved in a drug prescription and dispensation to prevent fraud") are defined in specification-level policies. Specification-level policies define control objectives, but they do not specify how the objectives can be achieved. This can be done in different ways: by blocking (e.g., forbid dispensation if the user that prescribed is the same trying to dispense medicine) or by executing additional actions (send a notification to an auditor requesting confirmation that the dispensation is not a fraud). The concrete choice is stipulated in so-called implementation-level policies, the focus of our framework.

Our framework can be used for specification and enforcement of implementation-level policies at different levels of abstraction. To illustrate this flexibility we describe an implementation of our framework at the OS and ESB levels. In this implementation, prescription and dispensation are services running in an ESB. Now, SoD policies enforced at the ESB level can be circumvented by a malicious user by directly modifying the files that contain the prescription and dispensation records, which motivates the need for OS-level protection. Moreover, our framework includes components that use trusted computing technology to create tamper-proof integrity measurement logs.

Research Problem: The problem we tackle is the design and implementation of a usage control framework that makes it possible to specify and enforce detective and preventive implementation-level usage control policies, that can be instantiated to different levels of abstraction, and that provides protection against tampering.

Contribution: We present such a system. In contrast to comparable approaches, our framework is based on a formal semantics, is more expressive because it supports temporal and cardinality operators, has been implemented and applied to real-world example scenarios, and provides protection against tampering with trusted computing technology.

Organization: This paper is organized as follows. §II introduces the language we adopt to specify usage control policies. §III describes the policy enforcement components and architecture. §IV introduces the guarantees we provide using trusted computing technology. §V presents two prototype implementations to enforce policies at the OS and ESB levels. §VI presents a performance evaluation. §VII positions our work with respect to related work, and §VIII concludes.

II. Specification of Usage Control Policies

In our framework we adopt an extension of the Obligation Specification Language (OSL) [2], [3] for the specification of usage control policies. The OSL formal semantics is defined over timed traces of sets of events that represent the execution of actions. Events have parameters and can represent actual and desired actions. Actual events correspond to the successful execution of an action, while desired events correspond to the tentative execution of an action (a request). This distinction allows the specification

of preventive policies that specify control actions to prevent actions and consequently policy violations from happening.

Policies in OSL are specified by means of detective and preventive mechanisms. Detective mechanisms consist of a condition and compensating actions that should be executed if the condition is satisfied. Preventive mechanisms exhibit an Event-Condition-Action (ECA) structure, and specify actions that are to be executed when the trigger event is observed and the condition is satisfied. The purpose of detective mechanisms is to observe and react while preventive mechanisms support the specification of enforcement policies that by design enforce authorizations and obligations.

The condition part of preventive and detective mechanisms is a set of action declarations and an OSL obligational formula (OSL formula for short). The set of action declarations is used to check the compliance of events in an OSL formula with their syntactic specification. In addition to event propositions, an OSL formula consists of XPath, attribute, propositional, temporal, and cardinality operators. The propositional operators *and*, *or*, *not*, *and*, and *implies* have the standard propositional semantics.

The $eventMatch$ ($action$, $isTry$, $*paramMatch$ ($name$, $value$)) operator returns true if the event corresponding to $action$ and with all matching parameters $name$=$value$ is observed in the current time step. Using the $isTry$ flag it is possible to specify matchings for tentative or actual events. The $attributeMatch$ ($subject$, $attribute$, $value$) operator compares the current value of the attribute identified by $attribute$ of a subject identified by $subject$ with $value$. This operator allow references to external information not present in the event when a formula is evaluated.

The temporal and cardinality operators are: $always$, $before$, $since$, $within$, $during$, $repSince$, $repMax$, and $repLim$. The semantics of these operators are described in previously published papers [2], [3]. The OSL operators supported in our framework are based on past time Linear Temporal Logics (LTL), but OSL also supports the future time counterparts (e.g. $since$ vs $until$). Our framework supports only past time because it is exponentially more succinct and operational than future time [4], [5], [6]. Moreover, it is often more intuitive to specify mechanisms considering that decisions can only be made based on knowledge of events that have already happened in the system. In this setting, the operators we introduced evaluate to true, false, or unknown at each moment in time. The unknown value is necessary because elements in XPath expressions may refer to missing elements, attribute values might not be available, or the observed past when temporal operators are evaluated may not be sufficient to decide the operator value.

The operator $xPathEval(expression)$ evaluates an XPath expression and is not part of the original OSL language. The $xPathEval$ operator increases the practical applicability of OSL because it allows arbitrary XPath expressions to be evaluated, including a rich set of numeric and string functions. XPath expressions are also supported in event matching operators or in actions of preventive mechanisms to refer to elements in the trigger event of preventive mechanisms (this is captured by variables in abstract OSL-based mechanisms).

Preventive mechanisms specify control actions that should be executed when the trigger of a mechanism is observed and the condition is satisfied. The trigger event of a preventive mechanism always references a tentative event. A control action states that a tentative action should be allowed, inhibited, modified, or delayed. To allow more flexibility, OSL supports configurable mechanisms using variables for event matching operators with unspecified action or parameter values. The following listing illustrates a configurable mechanism (a.k.a. *mechanism template*) for the SoD example using the OSL concrete syntax in XML. This template specifies that $dispensation$ is blocked if $prescription$ has been executed eventually in the past by the same $user$ for the same $prescriptionId$. These variables bind the mechanism to a specific scope, in this example $prescriptionId$ and $user$.

```
<mechanismTemplate name="SoDPolicy" type="preventive">
  <description>For all users: prevents prescription
    and dispensation by the same user.</description>
  <variable name="user"/>
  <variable name="prescriptionId"/>
  <trigger action="dispensation" isTry="true">
    <paramMatch name="user" value="#user"/>
    <paramMatch name="prescriptionId"
                value="#prescriptionId"/>
  </trigger>
  <condition>
    <not><always><not>
      <eventMatch action="prescription">
        <paramMatch name="user" value="#user"/>
        <paramMatch name="prescriptionId"
                    value="#prescriptionId"/>
      </eventMatch>
    </not></always></not>
  </condition>
  <inhibit/>
</mechanismTemplate>
```

It is known that it is non-trivial to generate efficient runtime monitors for parameterized specifications [7], which is the case of mechanisms templates. Templates are difficult to monitor because the number of parameter bindings can be very large and existing solutions to handle this problem are domain-specific. In our framework we specify deployment policies to support the efficient monitoring of mechanism templates. A deployment policy manages the *activation* and *deactivation* of mechanism template configurations. The template configuration references a template and instantiates the template configuration variables. By defining deployment policies we are able to efficiently manage the generation of runtime monitors for our mechanism templates.

III. Policy Enforcement Architecture

In our enforcement architecture business processes are monitored by a Policy Enforcement Point (PEP) that observes and intercepts action invocations taking into account event subscriptions of a Policy Decision Point (PDP). The PEP component signals these events to the PDP, and receives control actions (e.g. inhibit) in case a preventive mechanism

is triggered. The PEP component functions as a reference monitor for the business processes. PEPs are application-specific and can be implemented using runtime monitoring techniques such as OS system call interposition or at design time using aspect weavers in aspect-oriented programming.

Using the events signaled by the PEP, the PDP evaluates the active policies and retrieves attributes from an Attribute Resolver component. If actions are triggered as a result of the policy evaluation the execution of actions is delegated to an Action Resolver component. Action resolvers can be specialized components capable of executing complex business processes. The Attribute Resolver component essentially captures the behavior of Policy Information Points in the XACML framework [8].

The current set of policies deployed in the PDP component is retrieved from a Policy Repository component when a Policy Manager component signals the policy activation. The Policy Manager component includes an interface for authoring and management of policy deployment. Policies consist of preventive and reactive mechanisms, mechanisms templates, and deployment configurations encoded using the XML schema referred to in §II. In our architecture the PDP is a generic component that can be deployed in enforcement scenarios at different levels of abstraction. We have implemented this component in a C++ library that can be reused, provided that the XML policy language format is adopted. The main task when instantiating our architecture is to provide specific implementations of PEPs, Action Resolvers, and Attribute Resolvers.

The policy evaluation implementation in the PDP is an extension of the monitoring approach proposed by Havelund and Rosu [9] for past LTL formulas. LTL formulas can be efficiently evaluated considering the events from the current and previous time steps. Moreover, the approach does not require storage of past events, and is linear in the size of the formula when mechanisms are evaluated in each step.

Figure 1 shows the high-level behavior of our framework. The Policy Manager component stores a policy in the Policy Repository and receives an unique policy identifier (the policy's origin is not a subject of this paper). This identifier is used to activate the policy in the PDP component. The PDP component loads the policy from the repository and subscribes to the events in the respective PEPs. Whenever a PEP component observes an event it signals this event to the PDP that subscribed to it. The PDP evaluates the active policies and resolves attribute values if necessary.

After evaluating all active policies, triggered mechanisms may request the execution of actions in the Action Resolver component (if any), and return control actions (allow, inhibit, modify, or delay) to the PEP if preventive mechanisms are triggered. Control actions are returned only if a *tentative* event is signaled. If an *actual* event is signaled and a *detective* mechanisms is triggered the respective actions will be executed. We are aware that policies may be in conflict

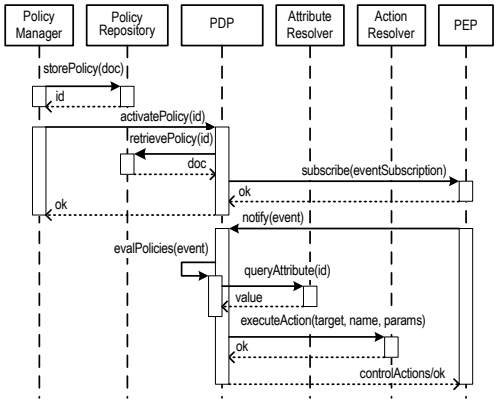

Figure 1. Policy enforcement sequence diagram

[10], but conflict handling is not yet supported by our framework.

IV. Authentication and Integrity Guarantees

Our usage control framework includes components for authentication, runtime integrity measurement, and remote attestation using trusted computing technology that is described elsewhere [11]. Every physical host running our policy enforcement components can benefit from these services.

The authentication is implemented using keys protected by the trusted platform module (TPM), and we adopt the Privacy Certificate Authority (Privacy CA) scheme. The Privacy CA schema allows the authentication of hosts using identity keys that are not directly bound to the TPM endorsement key. The objective is to support authentication without compromising privacy because unique identifiers are not used.

The runtime integrity measurement allows the content of files in the physical host to be checked using SHA-1 message digests. We use a kernel module to subscribe and receive file change events whenever the content of a monitored file is changed. In addition to the file content we also support the measurement of directory listings and output of commands. For security reasons a white listing mechanism is used to restrict the measurement of the commands to a limited set. In our framework we check the integrity of the policy enforcement components, check if the components are running using a ps command, and check the integrity of policies. The policy repository is simply the file system of the machine running the PDP component thus the checking of the policies integrity is comparably simple.

V. Implementation

We now present instances of our usage control enforcement framework for two levels of abstraction: the Enterprise Service Bus (ESB) and the operating system (OS). We have also implemented instances of our framework for other levels of abstraction, for example, web browser [12] and

the X11 window system [13]. Enforcement at different levels of abstraction is necessary to give additional security guarantees. The SoD policy we have introduced in §II refers to events associated to prescription and dispensation, which are services packaged in service assemblies that are deployed in an ESB. Each service assembly has its own data folder (working directory) where persistent service data is stored. The prescription and dispensation services store their records in files in their respective data folders. A malicious person willing to commit fraud can easily modify the prescriptions in these folders, dispense the medicine, and adjust the prescription and dispensation records later on so the fraud happens unnoticed. Therefore, policies are needed at the OS level to restrict modifications of the services' data folders only to the ESB process. Other options to commit fraud include modification of the ESB implementation or the deployment of malicious service assemblies that run under the same ESB process. Our ESB implementation guarantees by design that only the respective service assemblies have access to their data folders. Modifications of the ESB can be verified by us using integrity measurement support described in §IV.

ESB-level enforcement: In the ESB instance of our framework we used the open source Apache Servicemix ESB [14]. An ESB is a component responsible for routing service assemblies' business messages, implemented using different protocols, to support enterprise integration. Typical examples include messages exchanged between Web Services when a business process is executed. Enforcing policies at this level allows control over all business processes of an enterprise that are deployed in the ESB. This is possible because all messages exchanged by service assemblies flow through the ESB to be routed, independently from the protocol used between source and destination.

In an ESB, all messages sent by Business Services are handled by the Normalized Message Router (NMR, the core component of an ESB). These messages are transformed by the ESB into a normalized form, allowing a homogeneous and efficient inspection of their content. The customized components we have developed in this instance of our framework is the Enforcement Listener that at deployment time registers with the NMR and intercepts all the messages before they are routed. The NMR forwards the normalized messages to the Enforcement Listener component that extracts the data contained in the message and converts them in the event format in XML suitable for being processed by our PDP component. The PDP component evaluates the deployed policies and returns enforcement actions (e.g., allow and modify) to the Enforcement Listener.

If the message is allowed, it is released to the NMR for dispatch, otherwise it is discarded and an error message is returned to the original sender. The Enforcement Listener together with the specialized internal ESB components act as the PEP of our usage control framework. In this prototype the Enforcement Listener uses an external Action Resolver component to execute actions. Actions in the ESB scenario are directly mapped to calls for external Web Services that can be customized using parameters. Therefore, a mechanism specified in a policy is able to execute an arbitrary Web Service including the execution of a BPEL process. By adopting this instance of our framework the ESB users have additional guarantees and are able to enforce the SoD policy we introduced in §II.

OS-level enforcement: In the second instance of our framework we enforce usage control policies at the OS level in a host running OpenBSD. The customized components we have developed in this instance of our usage control framework are the Shell Wrapper and Syscall PEP. If a user logs in and starts a new shell the Shell Wrapper component is invoked instead of the standard user shell. The Shell Wrapper attaches its process to the Syscall PEP using a kernel module called Systrace [15]. After the Syscall PEP is attached to the process, it is able to intercept system calls executed by the process. If the user shell request the execution or receives the result of the execution of a system call, the Syscall PEP is notified, and an event is signaled to the PDP. The PDP evaluates the active policies and responds with an enforcement action or delegates the execution of an arbitrary action to a specialized action resolver. At the OS level, the action resolver component is able to execute standard OS commands. Considering our motivating example, this instance of our framework enforces policies at the OS level to restrict access to the service assemblies data folders only to the ESB process.

VI. Performance Evaluation

The performance of our policy enforcement components is influenced by different factors, for example: the event monitoring approach, the communication overhead between PDP and PEP (e.g. message encoding/decoding and network delay), the efficiency of the PDP component to evaluate the policies and respond to the PEP when events are signaled (event throughput), the context switching and synchronization overhead when components are deployed on the same core and compete for system resources, etc. Some of these performance factors depend on specific choices considering the application scenario and are out of the scope of this paper. In this section we discuss the event monitoring overhead for OS and ESB, and show event throughput performance results of our PDP component.

In the OS instance of our framework, the Syscall PEP only intercepts and notifies the PDP about events if subscriptions for the specific events have been previously made considering the deployed policies. The subscription ensures that system calls that are not referenced in the policies will not generate overhead. For system calls referenced in the policies Provos [15] shows that the overhead of intercepting system calls from a user space application increases the

execution time by 5 up to 30 percent depending on the system call and the type of arguments.

In the ESB instance of our framework the overhead to intercept an ESB message is the same for all business messages because the messages are already available in the ESB in a normalized format. We measured the response time of a SOAP service deployed in our ESB with and without the Enforcement Listener without considering the PDP response time. We observed that the overhead was a factor of 2 when the messages were intercepted and immediately allowed without sending event notifications to the PDP.

To evaluate the performance of our PDP component we have measured the response time when events are signaled by the PEP, which triggers an update of the state of all mechanisms whose condition references this event. We do not consider in our evaluation the time spent to resolve attribute values.

In our first experiment we have configured our PDP component to increase the number of deployed mechanisms, starting from 50 mechanisms, and incrementing 50 mechanisms per round until 1000 mechanisms were deployed. After every round of increment, we measured the real time to process 100 events in XML format. Each mechanism condition was a formula with 29 operators: 8 temporal and cardinality operators (one of each type), 10 event matching operators, and 11 propositional operators The events generated were not random, they were an exact match to the event referenced in the event matching operators in the condition of each mechanism. Figure 2 shows the event throughput when the number of mechanism instances increases when all 10 (solid line) and 5 (dotted line) event matching operators reference the signaled event.

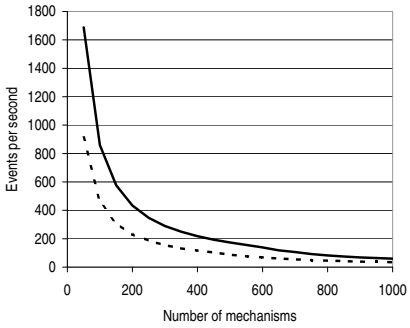

Figure 2. Performance Evaluation

Our graph shows that the event throughput decreases by a factor of 2 with the increase by the same factor of deployed mechanisms. Furthermore, the event throughput does not depend on the number of mechanisms, it depends on the number of event matching operators referencing the signaled event. The number of mechanisms does not affect the performance of our PDP because we implemented an internal index using a hash table for the event operators references. This index indicates precisely which formulas to update when an event arrives. With 10 thousand event references in the mechanism conditions for the exact same event the processing rate is around 34 events per second.

To verify the efficiency of our indexes we performed a second experiment, with 3000 mechanism instances using the same mechanism of our first experiment. However, 2000 mechanisms did not make reference to the signaled event and their objective was to introduce noise in the PDP evaluation. The performance results were almost identical confirming that we can efficiently process events and the dominant factor that affects the event throughput is the number of references in the formulas to the signaled event. In this second experiment, the total memory usage of our PDP with 3000 deployed mechanisms was around 55MB.

In our third experiment we measured the performance overhead of the SoDViolation mechanism introduced in §II. This mechanism consists of 2 event matching, 1 temporal, and 3 propositional operators. When 1000 SoDViolation mechanisms were deployed the event throughput for prescription and dispensation events matching exactly the references in the mechanisms conditions was 1800 events per second. We believe our performance results are promising considering that we do not expect so many mechanisms (e.g., 10 thousand) referencing the same event. However, many other aspects influence the performance and prevent us from giving a definite judgment without analyzing details of the application scenario. All experiments were performed on a machine equipped with an Intel Quad Q9650 3GHz CPU and 8GB of available system memory.

VII. Related Work

Existing frameworks for security policy specification focus mostly on authorization policies and do not support specification of obligations. Existing well-known academic and commercial general purpose usage control enforcement models and frameworks that support obligations are Ponder2, XACML, and UCON.

XACML [8] is an attribute-based policy language and enforcement framework that supports the specification of authorization policies and obligations. XACML has no formal semantics, supports grouping of policies, and provides meta constructs for conflict resolution. For example, a policy author can express that when more than one policy applies the first applicable is chosen or, if at least one policy evaluates to an allow decision, the access should be granted. Ponder2 [16] supports both authorizations and obligations by means of ECA rules. Ponder2 does not support temporal or cardinality operators, and conditions in ECA rules are limited to propositional expressions. Ponder2 supports four different types of events when an action is invoked, which capture the inbound and outbound request/response that are also supported by our framework. The UCON framework [17] specifies a family of models that supports complex

authorizations and obligations. This family of models addresses many different security aspects including rights, subject and object attributes, pre/ongoing/post attribute updates, and specialized policy conditions.

In contrast to our framework, XACML and Ponder2 have limited expressiveness because they do not support cardinality or temporal operators. UCON supports complex conditions, however, we are not aware of an efficient and practical implementation of the UCON framework. We consider other approaches to enforce usage control in specific application domains out of the scope of this paper because their objective is not a general purpose framework and they do not support policies as expressive as ours. We have applied state of the art techniques from the runtime monitoring domain in our framework to support efficient runtime monitoring of usage control policies [7]. In our case studies, the ESB Enforcement Listener is based on the work of Gheorghe et al. [18].

VIII. Conclusion

We have introduced a usage control framework for enforcement of usage control policies at different levels of abstraction. In our framework, the policy specification language is based on OSL, a formal language for usage control specification that supports temporal and cardinality operators. We extended OSL with deployment policies, XML query operators (XPath), and attributes. These extensions improve the practical applicability of the language and allow the implementation of efficient runtime monitors. Furthermore, our usage control framework also includes authentication and tamper detection using trusted computing technology. We are not aware of other frameworks that provide equivalent expressiveness, are efficient for runtime monitoring, and are integrated with trusted computing technology.

One assumption in this paper is that policies have to be specified independently for the different levels of abstraction. We are currently working on the integration of our monitors at different levels of abstraction, considering information flow policies.

Acknowledgment. This work was supported by FhG Internal Programs, Attract 692166, as well by the EU-funded IP MASTER.

References

[1] MASTER, "Managing assurance, security, and trust for services," Available at: http://www.master-fp7.eu.

[2] M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter, "A policy language for distributed usage control," in *Computer Security ESORICS 2007*, ser. Lecture Notes in Computer Science, 2007, vol. 4734.

[3] A. Pretschner, M. Hilty, D. Basin, C. Schaefer, and T. Walter, "Mechanisms for usage control," in *Proceedings of the 2008 ACM symposium on Information, computer and communications security (ASIACCS)*, 2008.

[4] R. Koymans, J. Vytopil, and W. P. de Roever, "Real-time programming and asynchronous message passing," in *Proceedings of the second annual ACM symposium on Principles of distributed computing (PODC)*. New York, NY, USA: ACM, 1983.

[5] F. Laroussinie, N. Markey, and P. Schnoebelen, "Temporal logic with forgettable past," in *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS)*. Washington, DC, USA: IEEE Computer Society, 2002.

[6] C. Dax, F. Klaedtke, and M. Lange, "On regular temporal logics with past,," in *Proceedings of the 36th Internatilonal Collogquium on Automata, Languages and Programming: Part II (ICALP)*. Berlin, Heidelberg: Springer-Verlag, 2009.

[7] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Rosu, "An overview of the mop runtime verification framework," *Int. J. Softw. Tools for Technol. Transf.*, 2011, available at: http://fsl.cs.uiuc.edu.

[8] E. Rissanen, "extensible access control markup language v3.0," Available at: http://docs.oasis-open.org, 2010.

[9] K. Havelund and G. Rosu, "Efficient monitoring of safety properties," *Int. J. Softw. Tools for Technol. Transf.*, vol. 6, Aug 2004.

[10] A. Pretschner, J. Ruesch, C. Schaefer, and T. Walter, "Formal analyses of usage control policies," in *International Conference on Availability, Reliability and Security (ARES)*, 2009.

[11] R. Neisse, D. Holling, and A. Pretschner, "Implementing trust in cloud infrastructures," in *11th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGrid 2011 (to appear)*, 2011, available at: http://zvi.ipd.kit.edu.

[12] P. Kumari, A. Pretschner, J. Peschla, and J. Kuhn, "Distributed data usage control for web applications: A social network implementation," *Proceedings 1st ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2011, available at: http://zvi.ipd.kit.edu.

[13] A. Pretschner, M. Buechler, M. Harvan, C. Schaefer, and T. Walter, "Usage control enforcement with data flow tracking for x11," in *Proc. 5th Intl. Workshop on Security and Trust Management*, 2009, pp. 124–137.

[14] "Apache servicemix," Available at: http://servicemix.apache.org, 2011.

[15] N. Provos, "Improving host security with system call policies," in *In Proceedings of the 12th Usenix Security Symposium*, 2003.

[16] K. Twidle, E. Lupu, N. Dulay, and M. Sloman, "Ponder2 - a policy environment for autonomous pervasive systems," in *IEEE Workshop on Policies for Distributed Systems and Network (POLICY)*, June 2008, pp. 245–246.

[17] J. Park and R. Sandhu, "The uconabc usage control model," *ACM Trans. Inf. Syst. Secur.*, 2004.

[18] G. Gheorghe, S. Neuhaus, and B. Crispo, "xesb: an enterprise service bus for access and usage control policy enforcement," in *In the proceedings of the IFIPTM10*, 2010.