Obligation Policies: An Enforcement Platform

Pedro Gama, Paulo Ferreira
INESC-ID/IST
Distributed Systems Group
Rua Alves Redol, n^{o}9, 1000-029 Lisboa
[pedro.gama, paulo.ferreira]@gsd.inesc-id.pt

Abstract

The use of policy-based mechanisms significantly reduces the complexity associated with applicational development and operation.

In particular, history-based policies allow the system to base application access decisions on the evaluation of other actions executed in the past. Obligation-based policies enhance this concept with the possibility of enforcing that certain actions will be executed in the future. This is a necessary evolution because some semantics are either easier to express as obligations or cannot be specified using traditional authorization mechanisms.

Currently, the absence of enforcement mechanisms for obligation-based policies imposes the implementation of ad-hoc functional constraints. This increases development time and introduces security vulnerabilities into the policy engine.

We present a policy platform called Heimdall,[1] *which supports the definition and enforcement of obligation-based policies. A prototype implementation is described, together with an evaluation which denotes encouraging results.*

1. Introduction

Several security constraints must be applied in runtime systems in order to control access to system resources. One approach for the definition of such constraints is based on the use of security policies[14]. These policies describe the access rules with an high level of abstraction, clearly separating the specification and the implementation of security mechanisms [33]. In particular, history-based policies allow the inclusion of temporal events in the policy rules. A common example of such type of policies is the Chinese-Wall security policy [4], which analyzes previous actions in order to authorize (or not) the current operation.

In addition to the possibility of including past events, a policy can specify that certain operations must be executed in the future, forming an obligation policy.

Several examples can be given which evidence the relevance of obligation policies:

- A user pays for a QoS (Quality of Service) agreement in a grid node[13]. The node is obliged to provide 5 hours of CPU-time in the next 24 hours.
- A contributor submits his tax declaration and is obliged to send the receipt to its employer within 7 days.
- An online merchant issues 5 euros bonus vouchers for his new clients. If a client uses the voucher he is obliged to purchase a global amount of 50 euros during the voucher validity period. It is important to notice we are assuming a new more flexible semantics not usually found in voucher systems. Usually a merchant issues a voucher valid for a period of time and a minimum individual purchase amount. This limits the flexibility of the voucher and consequently the user's purchases. Alternatively, we allow the voucher to be checked at any purchase, but the user is obliged to shop a minimum specified amount in the validity period.

The application of obligation policies is the natural solution for the enforcement of the scenarios presented above. Although it is sometimes possible to transform an obligation policy into an authorization policy (i.e. based solely on past events), the existing mechanisms for that conversion cannot be used in these cases [26]. The fact that the obliged action (e.g. guarantee a QoS) is causally dependent on the trigger action (e.g. pay a QoS guarantee) implies that the actions cannot be reversed in order to transform the obligation policy into a history-based one:

- In the first example, it isn't possible for the user to pay the QoS guarantees after the resource usage, as the usage itself depends on a previous reservation.
- In the IRS case, the contributor cannot send the receipt to its employer before it is generated upon tax submission.
- Finally, in the voucher case, a policy engine cannot deny a purchase based in the validity of the voucher or

[1]Heimdall is the watchman of the gods in Norse Mythology. He possesses a "second sight" that allows him to see into the future.

on previous purchases, as the user can still fulfill the obligation in the future.

It is obvious that obligation policies can be implemented programmatically together with other functional constraints. This is the common situation today, due to a lack of expressiveness and enforcement capabilities in current policy platforms [27, 8, 18]. However, this approach leads to the possibility of severe security errors in the policy platform. In addition, it generates the need to modify and redeploy the application whenever the policy changes.

Several systems were proposed to support obligation policies. Most focus primarily on the definition of the obligation model rather than in the enforcement of the policy itself [27, 30, 3, 12, 28]. Some effectively enforce obligation policies, but their approach is either based in ECA (event-condition-action) mechanisms[8], making them more appropriate for workflow processes, or the platform is focused for a particular obligation enforcement, like resource provisioning [32].

Such enforcement is indeed a complex task. As pointed out by Schneider [29], it is not possible to base an authorization decision based on future actions through the use of a standard execution monitor. This is due to the fact that the execution monitor would have to analyze all possible future sequence of events, something only attainable with a static analyzer. However, Schneider bases his statements in the fact that every event analyzed for authorization is independent from all the other events. Ribeiro [26] suggests that in different execution environments, such as those found in transactional engines, several distinct events can be interrelated in an atomic way. Thus, if action A obliges action B, and B is not executed, we can invalidate A, instead of ”obliging” B. In practical situations however, we find this invalidation to be usually impossible, due to the fact that either an action can't be cancelled, or has generated secondary effects after execution.

Thus, we base our alternative approach on the hypothesis that any executed action can be counterbalanced in the future. In this context, although a policy monitor cannot enforce the execution of a future action, it can compensate any previously executed actions. For example, in the QoS case described above, if the grid node doesn't comply with the QoS agreement, it might be faced with public exposure in a blacklist server until he refunds the client.

The platform presented in this paper, called Heimdall, provides a transparent[2] enforcement mechanism for obligation policies. This allows policy administrators to define obligation policies independently of application development. The policies themselves are enforced without the need for further application integration. To our knowledge this is the first pratical design, implementation and evaluation of a platform that supports the specification and enforcement of history-based security policies and, more importantly, obligation policies. Furthermore, this platform can be integrated with other runtime systems to extend their security capabilities.

[2]Transparency means in this context that application development is independent from policy specification and enforcement.

This paper is organized as follows. In the next section we present a general overview of Heimdall, describing a generic application execution. In Section 3 we define our policy language: the xSPL. We then present the architecture of the system in Section 4 describing the various components of the platform. In Section 5 we focus on the prototype implementation, further evaluated in Section 6. Then, we contextualize Heimdall within this scientific field discussing some related work in Section 7. Finally, in Section 8 we present our conclusions.

2. System Overview

Heimdall is a Middleware platform with the architecture depicted in Figure 1.

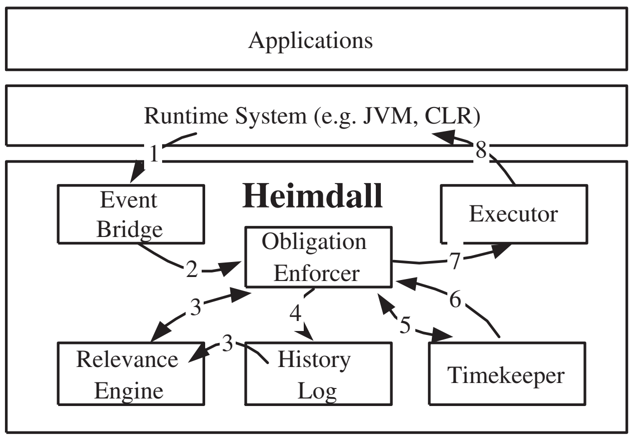


Figure 1. Overview of Heimdall.

It was developed with three major objectives in mind:

- Offer a comprehensive set of language semantics for policy definition. This includes obligation, history-based and RBAC policies.
- Clearly separate policy specification, enforcement and application development.
- Provide easy integration with existing runtime systems and applications.

In what concerns the first goal, Heimdall extends the SPL (Security Policy Language)[27] in order to increase its expressiveness with necessary obligation semantics.

Separation of application development and policy specification and enforcement is also a key target in Heimdall. The runtime system only has to make sure that a notification of every relevant applicational operation (called an event) is generated upon execution. The policy administrator will then define the obligation policies based on these event descriptions. This approach allows the integration of events from different applications into a common policy without the need for additional application development. For the execution of compensatory actions, the policy administrators can flexibly define a callback method for invocation by Heimdall (e.g. a webservice method).

Seamless integration with runtime systems (e.g. the Java Virtual Machine or the .Net Common Language Runtime) is provided through Heimdall interface modules, more specifically the Event Bridge and the Executor. The input to Heimdall can be easily generated by intercepting security critical operations in the runtime system and composing associated events. On the other hand, the invocation of compensatory actions can be accomplished without runtime system modifications through the use of webservices or other existing protocols. This architecture allows any runtime system to use our obligation policy engine with minor integration efforts. Additionally, Heimdall can coexist with other policy and/or security engines without any interference.

For clarity we illustrate in Figure 1 the execution of a generic application over a runtime system which uses Heimdall mechanisms to enforce obligation policies.

1. The runtime system sends an operation description to Heimdall. Notice that Heimdall doesn't block the operation in the runtime system, which proceeds its normal execution. Our obligation semantics specifies that an action is always authorized, with potential compensatory actions executed in the future.
2. The operation description is received in Heimdall by the Event Bridge module. This module translates the operation description into a normalized construct: the xSPL event. This event is then sent to the Obligation Enforcer.
3. The Obligation Enforcer contacts the Relevance Engine in order to determine if the current event is applicable for any of the existing policies. If any of the policies is history-based, the Relevance Engine retrieves applicable past events from the History Log in order to analyze the evolution of the policy.
4. If the event is applicable for policy monitoring, it must be stored for future evaluations in the History Log.
5. Additionally, if the current event triggers a new obligation, the Obligation Enforcer gets a new obligation timer from the Timekeeper module, and associates it with a new obligation instance. On the other hand, if an obligation is fullfilled, the Timekeeper is contacted to cancel an existing timer.
6. Finally, in the case an obligation timer expires before the corresponding obligation is fulfilled, the Timekeeper informs the Obligation Enforcer about the fact.
7. The Obligation Enforcer invokes the Executor in order to execute the compensatory actions specified in the policy.
8. The Executor serves as the interface between Heimdall and the runtime system in what concerns the execution of compensatory actions. This can be achieved using a number of protocols (e.g. webservices, HTTP, RMI, etc), providing a flexible callback interface to client applications.

3. xSPL

The obligation policies are defined in xSPL (eXtended Security Policy Language), an extension of the SPL language [27]. This language allows the definition of policies with complex constraints, including history-based and obligation-based policies.

Each policy is defined by the composition of several rules. The rule is formed by two distinct sections:

$$< trigger\ expression :: obliged\ expression >$$

The trigger expression specifies the applicability of the rule. If this expression is true it means a new obligation must be enforced. On the other hand, the obliged expression defines the conditions which fulfills an obligation.

One of the central concepts in xSPL is that of the event. Each action in the system, like the payment of a QoS agreement or the monitoring of the simulation CPU time, is mapped into a normalized event construct. These events can be instantiated in two different sets. The Past Events Set includes the events which were previously executed. The Future Events Set specify actions intended to be executed in the future (like the Simulation Finished event). Additionally, the event containing the operation being evaluated is called the current event (represented as 'ce').

For clarity we present in Figure 2 a definition in xSPL of the discussed QoS agreement policy (c.f. Section 1).

```
?QoSPolicy ():
  TIMEOUT = 24 hour
  COMPENSATE = blacklist(ce.target)
  EXIST fe IN FutureEvents {
    ce.action = "Pay␣QoS"
    ::
    (fe.action = "Simulation␣CPU␣time" AND
    fe.action.parameter[0] >= 5 hour) OR
    fe.action = "Simulation␣finished"
  }
}
```

Figure 2. Example of a QoS obligation policy in xSPL (note that the parameter[0] of the "Simulation CPU time" event contains the Simulation CPU time).

This policy states that if a user pays for a QoS agreement, the grid node is obliged to provide at least 5 hours of CPU time to a simulation process, before the obligation expires in 24 hours. However, if the simulation finishes before that time, the obligation is also considered to be fulfilled. If the QoS obligation is not fulfilled, Heimdall will automatically invoke the blacklist service. We can imagine the offending node is removed from the blacklist in the future against the refund of the QoS agreement, but that is out of the scope of this paper.

It is important to define the policies with care in order to assure the correct semantics is implemented. Some subtle differences can modify completely the objective of the policy. Consider the example policy of Figure 3. At first sight one might read it as ”Whenever someone buys a book, someone must pay for the book”. That is not correct. The exact meaning of the policy is that ”If someone buys a book, someone has to pay for a book”. The book might not be the same, leading to a situation in which several books are bought, and only one is paid.

```
?SimplePolicy ():
  EXIST fe IN FutureEvents {
    ce.action = "Buy_book" :: fe.action = "Pay_book"
  }
}
```

Figure 3. An obligation policy not instantiated.

In order to define the desired semantics one would have to properly instantiate the executed actions, as shown in Figure 4.

```
?SimplePolicyInstantiated ():
  EXIST fe IN FutureEvents {
    ce.action = "Buy_book"
    ::
    fe.action = "Pay_book" &
    fe.action.parameter[0] = ce.action.parameter[0]
  }
}
```

Figure 4. An obligation policy properly instantiated (note that the parameter[0] of each action contains the identifier of the book).

In this example we oblige both actions to be performed on the same book. For simplicity we didn't instantiate the simulation in the QoS example, but the process would be similar to this case.

Two other concepts introduced in xSPL are those of the timeout period and the compensatory actions. The specification of the timeout period (tag TIMEOUT) relates to the time allowed to fulfill an obligation. This element is crucial, as the timeframe is normally different depending on the specific policy (e.g. one hour might be a reasonable timeout for an online payment, but is probably too short for an asynchronous email reply).

The definition of compensatory actions (tag COMPENSATE) is also a key issue in what concerns the enforcement of obligation policies. Such compensatory actions should either be sufficiently penalizing so that one is ”forced” to fulfill the obligations or, when possible, cancel already executed actions.[3]

Note that two reasons usually prevent the effectiveness of the latter approach (i.e. cancellation of previous actions):

- First, the cancellation of real actions is not possible (e.g. a system cannot cancel a printout it has already generated).
- Additionally, this cancellation, when possible, might not be sufficient to counterbalance secondary effects of the executed actions (e.g. the IRS department might cancel a contributor's submission, but that is hardly enough if a tax refund was already executed).

[3]By cancelling an action we mean to completely erase the effects of the action, similar to what happens when a database transaction is aborted.

Those are the reasons why compensatory actions should be as penalizing as possible, in order to be unprofitable not to fulfill an obligation.

4. Architecture

Heimdall encloses six functional modules, as already depicted in Figure 1.

The Event Bridge is the input processing module of Heimdall. It plays an important part in the interoperability of the platform with different runtime systems. The most basic Event Bridge merely receives events and forwards them to the Obligation Enforcer. These events (e.g. a file access request or a payment notification) are generated by the runtime systems, upon the execution of operations by applications. New bridges can be easily developed which translate proprietary operation definitions into standard xSPL events.

The Executor module on the other hand calls back the runtime system (or any other platform defined in the compensatory actions) in order to compensate not-fulfilled obligations. Its flexibility (webservices, HTTP, RMI, etc) allow an application to combine obligation policies with compensatory actions in a penalizing and effective manner.

The Obligation Enforcer, together with the Relevance Engine, are the core of the Heimdall platform. They are described below in more detail.

The Timekeeper provides fulfillment timers, being used to control the expiration of an obligation instance.

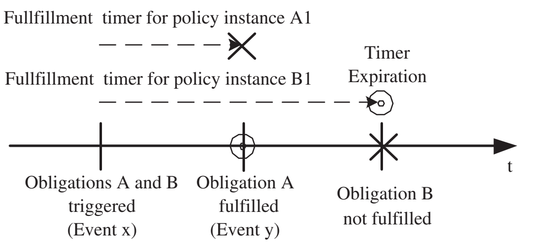
Finally, the History Log filters and stores the events which are relevant for policy evaluation.

4.1 Obligation Enforcer

The Obligation Enforcer is the coordinator for overall obligation enforcement. It ensures the system is in a consistent state with respect to obligations by compensating obligation instances which have not been fulfilled in due time.

This module is triggered in two different situations: the issuance of an xSPL event by the Event Bridge (upon the execution of an operation by an application) and the timeout of an obligation fulfillment period. In order to illustrate these mechanisms we use the generic execution illustrated in Figure 5.

When the event x is received, the Obligation Enforcer contacts the Relevance Engine. This module indicates that the current event (x) triggers the obligation policies A and B. Thus the Obligation Enforcer creates two new obligation instances (A1 and B1), associating them with fulfillment timers obtained from the Timekeeper module.



Fullfillment timer for policy instance A1
Fullfillment timer for policy instance B1
Timer Expiration
t
Obligations A and B triggered (Event x)
Obligation A fulfilled (Event y)
Obligation B not fulfilled

Figure 5. The evolution of two simultaneous obligations.

Afterwards, the Event Bridge sends the event y to the Obligation Enforcer. After evaluation of the event, the Relevance Engine indicates that the instance A1 of obligation A is fulfilled. Therefore the Obligation Enforcer stops the associated fulfillment timer and deletes the obligation instance.

Finally, when the fulfillment timer of obligation instance B1 expires, the Timekeeper Module alerts the Obligation Enforcer. The Obligation Enforcer sends the instance information to the Executor module, in order to execute compensatory actions. Additionally, it deletes the obligation instance, as no further actions are necessary.[4]

4.2 Relevance Engine

The mechanism associated with the Relevance Engine basically evaluates a certain policy against the current event and the content of the History Log. This allows the monitoring of policies in which the trigger and the obliged expressions (c.f. Section 3) only refer to the current event and to past events, such as the case of history-based policies. Following evaluation, the mechanism returns one of three possible results:

- It returns positive if the trigger and obliged expressions are true.
- It returns negative if only the trigger expression is true.
- It returns not-apply if the trigger expression is false.

The History Log cannot obviously contain events that will occur in the future. This prevents the correct evaluation of the obliged expression in the case of obligation policies. It is important to note that two milestones are critical for the enforcement of obligation policies: the triggering of the policy itself, and the fulfillment of a specific policy instance. Therefore, in order to use the mechanism mentioned above, Heimdall performs an ”aging” operation in obligation policies: future events are instantiated as current events in order to postpone evaluation until the moment they are executed. For clarity, consider the obligation policy presented in Figure 6.

[4]An enhancement to this alternative would be to consider the replacement of a not fulfilled obligation by a new obligation with stricter rules, as proposed in [12]. We plan to introduce this possibility in future versions of Heimdall.

```
?Door:
  EXIST fe IN FutureEvents {
    ce.action = "Open_Door" :: fe.action = "Close_Door"
  }
}
```

Figure 6. An obligation policy.

We internally separate each obligation policy into two distinct policies, individualizing the evaluation of obligation triggering and fulfillment. The result is presented below in Figure 7.

The DoorTrigger policy indicates that the Door policy is triggered by the current event: a new instance must be created and associated with fulfillment timers. On the other hand, the DoorFulfill policy indicates that a certain instance of the Door policy is fulfilled by the current event. In this way, obligation policies are monitored by the same mechanisms used for history-based policies, minimizing the complexity of the reasoning engine.

```
?DoorTrigger:
  ce.action = "Open_Door" :: true;

?Doorfulfill:
  true :: ce.action = "Close_Door"
}
```

Figure 7. Internal separation of the Door policy.

Other evaluation issues are related with the semantics associated with generated events. In Heimdall one single event can be associated with several obligation policies, as already illustrated in Figure 5 above. In this example, obligations A and B are triggered by the occurrence of the same event (x). This is a common situation in multi-application systems (e.g. an IRS submission might oblige a user not only to send the receipt to its employer but also to pay any existing debt). It also provides a more flexible way for policy administrators to specify the system policies, as they are not restricted to map each event to a specific obligation.

In Heimdall, distinct obligation policies triggered by the same events execute independently from one another. This means that one of them could be fulfilled while other is not, leading to the execution of compensatory actions. One could argue that if an action is compensated, it should not be involved in a fulfilled obligation. However we consider the desired semantics to be rather application-dependent, and so in Heimdall it is the responsibility of the policy administrator to resolve any conflict related to not-fulfilled obligations. In other words, if the execution of compensatory actions affects the consistency of other obligations, the compensatory actions themselves must restore the overall consistency of the system (e.g. by also compensating other obligations which were already executed).

5. Implementation

Heimdall was implemented in Java and tested over the Java Runtime Engine 1.4.2.

The Event Bridge receives events from any runtime system installed in the same node as Heimdall and forwards them (the events) to the Obligation Enforcer. At present time the events must be received in the xSPL format. However, we are currently incorporating a XML-to-xSPL convertor into the Event Bridge, in order to allow the processing of XML-based events, thus enhancing platform compatibility.

The Relevance Engine adapts and extends the functionalities of the reasoning engine proposed by Ribeiro [27] in order to allow the monitoring of obligation policies.

The Obligation Enforcer processes incoming events and activates compensatory actions. For the first goal it manages two repositories, implemented as normal Java Vectors. One with a descriptor for each policy deployed in the system, and another with all pending obligation instances. Whenever an event is received, all these descriptors are sequentially transmitted to the Relevance Engine, in order to evaluate if the current event triggers a policy or fulfills an obligation instance.

In what concerns the execution of compensatory actions, the Obligation Enforcer possesses a distinct thread of execution which is activated by the Timekeeper module in case a timer expires.

The History Log is implemented as a set of event repositories, one for each of the policies in the system. At policy deployment time, each of these repositories is dynamically generated according to policy definitions, in order to optimize event storage and retrieval. A filtering mechanism discards any event that is not relevant for the evaluation of the policy (e.g. if a policy controls a simulation QoS, it doesn't need to store events related to file access). In addition, the repository stores only useful evaluation fields (e.g. if a policy controls the number of books sold, it doesn't have to store the price of the books). This approach allows us to minimize Log dimension, and thus enhance performance of event retrieval (for more details see [27]).

Finally the Executor Module parses and invokes the defined compensatory actions. The current implementation only provides HTTP-based callbacks (e.g. *http://myserver/compensate.jsp?id=1*), but new interface modes are being developed and will be available shortly.

6. Evaluation

Performance is critical for any policy engine, as it should not interfere with the normal execution of operations. It's worthy to note that, in Heimdall, the runtime system (and consequently the application) never blocks waiting for the evaluation of obligation policies.

In order to assess the scalability of the platform, it is mandatory to evaluate Heimdall mechanisms in terms of the time it takes to evaluate an event received from the runtime system.

Heimdall performance is closely related to the cost of checking the relevance of a certain event in terms of the policies deployed in the system. This includes analyzing if any of the policies is triggered by the current event, and further checking if any of the pending obligation instances can be fulfilled by that event. Thus, two major aspects influence the relevance checking: the number of events in the History Log and the number of simultaneous pending obligations.

The effect of executing compensatory actions has minor consequences in what concerns the overall capacity of the system. The number of compensatory actions executed is normally one or more orders of magnitude smaller than the number of generated events. Furthermore, the associated processing only involves the invocation of a specified action in the runtime system. For these reasons, and for lack of space in this paper, we don't present an evaluation on the effects of compensatory actions to the overall performance of the system.

We further focus the evaluation in what concerns obligation policies, as obligation enforcement is the main goal of this paper (Heimdall also supports other types of policies like RBAC).

With these goals in mind, we developed and executed two different test scenarios in order to evaluate the influence of the mentioned aspects to the behavior of the system. Both scenarios were tested in a Pentium 4, 2.8GHz, 512MB PC, running Microsoft Windows XP Professional SP2.

6.1 Strict Obligation Policies

By strict obligation policies we mean those obligation policies which are not history-based. This distinction is important, as simulations show significant performance differences between these two types of obligation policies.

Our first test scenario tests Heimdall with a set of similar strict obligation policies, presented in Figure 8.

```
?StrictTestScenario:
  EXIST fe IN FutureEvents {
    ce.action = "An" :: fe.action = "Bn"
  }
}
```

Figure 8. A strict obligation test scenario (An and Bn are instantiated as A00, B00, A01, B01... for each of the specific policies).

It expresses that after executing the action "An", the entity is obliged to perform the action "Bn" in the future. The results of the simulation are presented in Figure 9.

The lines in the chart correspond to different simulations with a distinct number of policies in the system (from 1 to 100 policies). During the simulations we generate events in order to sequentially trigger all policies, fulfill all policies and start the cycle again. As a result, the number of simultaneous obligation policy instances is equal, in average, to half the number of deployed policies. The values in the chart were obtained by averaging the evaluation time in 5,000 events intervals. In this and subsequent charts each average is presented in the end of the associated interval

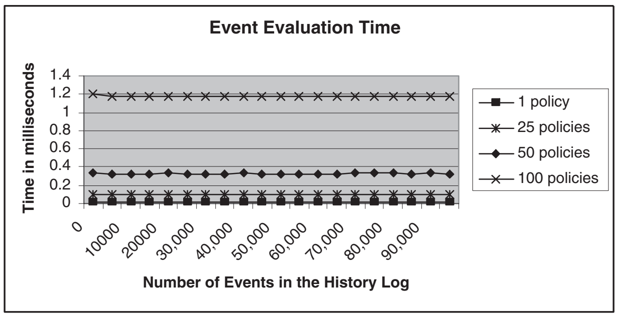


Figure 9. Event Evaluation Time with simple obligation policies.

(e.g. the average evaluation time in the 25,000-30,000 interval is presented in the 30,000 index). Notice that in this particular case the number of generated events is similar to the number of events in the History Log, because all events are relevant to the evaluation of policies, and so must be stored for future analysis.

This chart shows that in what concerns the policy presented in Figure 8, the number of events in the History Log is not relevant for event evaluation time. This is due to the fact that Heimdall only has to analyze the current event in order to check policy triggering and fulfillment (c.f. Relevance Engine in Section 4). The differences in evaluation time between simulations with different number of policies are due to the necessity of checking if the current event is a trigger for each of the deployed policies.

Afterwards, and using the same test scenario we continually generated trigger events, in order to increase the number of simultaneous obligation instances. The results of the simulation are presented in Figure 10.

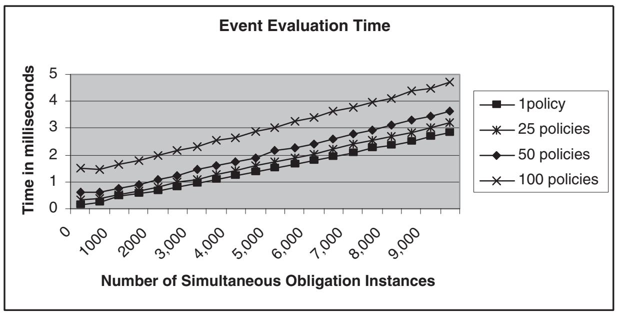


Figure 10. Event Evaluation Time with several obligation instances.

The lines in the chart correspond to different simulations with a distinct number of policies in the system (from 1 to 100 policies). As in the example above, the values were obtained by averaging the evaluation time in 5,000 events intervals. In these simulations the number of generated events is similar to the number of pending obligation instances, as we only generate trigger events, and thus never fulfill obligations (the expiration period is set to an high value in order not to interfere with the evaluations).

As expected, the evaluation time increases with the number of pending instances, as each of them must be checked for fulfillment.

However, the performance of Heimdall is rather acceptable with this type of strict obligation policies. The evaluation time with 100 strict obligation policies deployed in the system ranges from 1 millisecond to 4 milliseconds, depending on the number of simultaneous obligation instances, ranging from 50 (in the first chart) to 9,000 (in the second chart).

If generated events don't trigger any obligation policy, Heimdall obviously doesn't create any obligation instance. Additional simulations show that in this case the event evaluation times decrease in about one order of magnitude.

6.2 History-Based Obligation Policies

Heimdall also supports the inclusion of history-based rules in obligation policies. Although an alternative formulation for the required semantics can be achieved by combining distinct history-based and obligation policies, this approach provides a more natural way to express certain semantics in several practical situations. Thus we developed a second test scenario with the history-based obligation policy presented in Figure 11.

```
?SecondTestScenario:
  EXIST pe IN PastEvents {
    EXIST fe IN FutureEvents {
      ce.action.name = "An"
      ::
      pe.action.name = "Bn" & fe.action.name = "Cn"
    }
  };
```

Figure 11. An history-based obligation test scenario (An, Bn and Cn are instantiated as A00, B00, C00, A01, B01, C01, ... for each of the specific policies).

It expresses that after executing the action "An", the entity is obliged to perform the action "Cn" in the future. Additionally, at fulfillment time, action "Bn" must have already been executed (prior to the execution of "Cn"). The results of a simulation in which new obligation instances are continually triggered is presented in Figure 12. Note that obligation instances are never fulfilled in this scenario.

The lines in the chart correspond to different simulations with a distinct number of policies in the system (from 1 to 25 policies). The values were obtained by averaging the evaluation time in 50 event intervals.

The chart shows that event evaluation times are one order of magnitude greater than the ones in the scenario of Figure 10. This is due to the fact that for each generated event, Heimdall must search the entire log in order to check if any of the pending obligation instances can be fulfilled. Note that in the simulations presented in Figure 10 the Rel-

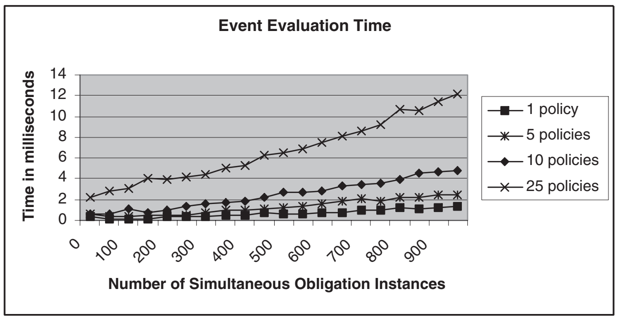


Figure 12. Event Evaluation Time with several obligation instances in a history-based obligation policy.

evance Engine didn't have to search the Log, as only strict obligation policies were deployed. The evaluation times are however still reasonable, as a system with 25 installed policies and 500 pending obligation instances takes about 6 milliseconds to evaluate one event.

In addition, together with the first scenario, this result demonstrate that the system scales efficiently with the increase in the number of events in the History Log and the number of simultaneous pending obligations.

7. Related Work

The concept of obligation was already proposed and discussed in several papers, with details and even semantics differing significantly.

Minsky [21] introduces several notions related to obligations, like the fulfillment time and the compensatory actions associated with not-fulfilled policies. The proposed system is not intended for policy enforcement though, but rather to control transitional violations of system integrity. The LGI (Law-governed interaction) platform[22] extends these concepts by enforcing the policy of the system (named Law) through the exchange of messages between controllers. The rules of the policy are specified directly in Prolog and define a message workflow. Therefore, the system is more focused in the coordination of distributed applications than in the enforcement of security policies. LGI can control history-based and obligation policies, but the management (storage and retrieval) of the events must be programmed into the policy definition, which difficults the implementation of complex policies. Heimdall, on the other hand, automatically generates the necessary instructions from the policy definition. The specification of the compensatory actions in Prolog also limits the flexibility of obligations. Alternatively, Heimdall allows connections to webservices, web applications and other protocols. Finally, the efficiency evaluation of LGI is centered in the message exchange between controllers, disregarding scalability issues related to concurrent and complex obligation policies.

Ribeiro [27] defines a security policy language which includes the concepts of history-based and obligation policies. He proposes a mechanism for the enforcement of obligation policies [26] but requires the issuance of an external "commit" event. When this event is generated the reasoning engine verifies if the obligation policy is fulfilled. Furthermore, the mechanism doesn't allow the instantiation of a particular obligation instance, and so the system can only determine if all obligation instances were fulfilled. Heimdall allows the enforcement of security policies without any additional events (apart from the ones corresponding to actions in the runtime system). Each obligation instance is individualized and thus can be checked for fulfillment apart from the others. In addition, Heimdall interfaces with runtime systems enhancing existing security mechanisms.

Policies are also being increasingly used to specify the rights and duties (obligations) of one organization in regard to other entities[31, 9, 20, 30, 15, 1]. This kind of policies is referred to as contracts and harmonize the interactions between different organizations. Specifically, in grid and peer-to-peer environments[13], they allow organizations to securely supply their resources against a defined retribution.

Chiu[6] presents a meta-model for the definition of contracts, including the concepts of obligation, permission and prohibition. The specification of the contracts uses a UML graphical notation, which can make the definition of a contract a complex task. Obligations are enforced using a ECA (event-condition-action) approach. This means that after the occurrence of a certain event the system checks if a specified condition is valid. In the affirmative case the action in the ECA rule is executed. These rules are enhanced with clock timers in order to provide a fulfillment period for obligations. A proof-of-concept prototype is not presented, difficulting the analysis of the system's behavior in practical situations.

Ponder [8, 10] and PDL[18] also use the event-condition-action concept in order to enforce obligations. However, their concept of obligations differs from the one in Heimdall, as the obliged actions are initiated immediately upon the occurrence of the trigger event. This makes the systems more appropriate for a workflow engine than for generic obligation enforcement. Additionally, they don't specify any compensatory actions regarding failure in complying with the policy.

Firozabadi[11] defines a contract as the policy that conducts the interaction between different entities. Each contract specifies the obligations and entitlements which are associated with a certain entity. While obligations express future conditions that must be fulfilled (e.g. provide storage capacity), entitlements prove that a certain entity has the right to access a certain resource. The obligation notion used in this system is less expressive than that in Heimdall, as obligations are effective in a specified timeframe (e.g. provide 300GB of storage from 15h00 to 20h00), rather than being activated by a particular event. In order to enhance the flexibility of contracts in what regards obligation fulfillment, this system introduces the notion of a contract block. This consists in a chain of cost-increasing obligations that provide alternative fulfillment courses for obligations. Heimdall policy language also allows this semantics through the disjunction of alternative policy rules[27]. In addition, Heimdall allows the monitoring of the specified contracts, together with the enforcement of not-fulfilled obligations through compensatory actions.

Wasson [32] proposes a system to support resource provisioning in virtual organizations (VOs)[13]. The system is centered around the concept of a Grid Bank which provides users with credits to use the VO'resources. Although including different components for policy monitoring, accounting and enforcement, it is focused on controlling just three types of policies involving provisioning, which limits its applicability. Heimdall on the other hand proposes a more general approach in which compensatory actions can be flexibly defined by the policy administrator.

Deontic Logic is used in several systems that support the notion of obligations. This type of logic extends the notion of an ideal system behavior with the possibility to analyze allowed deviations to this ideal conditions [17, 7]. However, the existence of some paradoxes difficult the application of this logic to policy systems [19]. We emphasize the importance of the contrary-to-duty paradox [25, 5], in which a secondary obligation is activated by the failure to fulfill a primary obligation. This concept is similar to the notion of compensatory actions in Heimdall, but extremely difficult to reason about in deontic logic systems. Nevertheless Kent [17] shows how temporal constraints and error recovery actions can be used in these systems.

Milosevic [20] and Molina-Jimenez [23] propose the use of finite-state machines to monitor the fulfillment of e-contracts. In the first case, the possible states are divided into 5 different levels, ranging from full compliance with the contract to global failure of the obliged actions. In the failure states, compensatory actions are in order, but no details are provided in what concerns the enforcement of their execution. In the latter paper, the authors explicitly penalize a failing entity by restricting the actions it can execute in a certain state, but no further compensatory actions are supported. In both systems, a performance evaluation is lacking in what concerns systems with complex policies, possibly leading to large finite-state machines.

Another related topic is that of provisions. These are actions that must be executed prior to a authorization decision. Jajodia [16] proposes the use of provisional authorizations to enhance traditional authorizations mechanisms. This allows the resolution of situations in which the access to a resource is currently denied, but the execution of a certain action might change this evaluation. Thus, the proposed mechanism presents the user with a number of alternative actions that can be executed in order to grant access. Bettini[3] extends this system with obligations. It formalizes both concepts (provisions and obligations) and proposes a system to automatically select a choice among alternative provisions and obligations. Obligation monitoring in this system is discussed in [2], introducing compensatory actions and time triggers. However, the system doesn't support history-based policies as Heimdall. Furthermore, no prototype evaluation was presented in order to demonstrate the feasibility of the system.

Park and Sandhu formalize obligations in the context of a usage control model[24, 34]. However the concept of obligations that is discussed involves the execution of actions prior to a certain authorization decision, which effectively ressembles more the concept of provisions. The notion of obligation used in Heimdall is referred to as long-term obligations.

8. Conclusions

In this paper we evidenced the relevance of obligation policies. We showed several examples in which the required usage semantics cannot be defined using traditional authorization mechanisms, or even history-based policies. The absence of effective enforcement mechanisms for this kind of policies (i.e. obligations) compels ad-hoc implementation of security constraints. This situation leads to security vulnerabilities.

We developed a prototype of the Heimdall platform, which allows the specification and enforcement of obligation-based policies. In addition, it also supports other kind of policies like history-based and RBAC. This platform can be easily integrated with existing runtime systems through its interface modules, as shown in the architecture description.

Finally, we found the performance penalization associated with Heimdall operation to be negligible. Runtime systems only have to generate event descriptions upon the execution of any security critical action. The Heimdall platform doesn't block the execution of the action which can proceed in parallel. Our evaluations further show that the performance of the Heimdall platform in terms of event processing is adequate for practical situations, efficiently scaling for a large number of policies and thousands of pending obligations instances.

References

[1] P. Ashley, M. Schunter, and C. Powers. From privacy promises to privacy management: a new approach for enforcing privacy throughout an enterprise. In *Proceedings of the ACM New Security Paradigms Workshop (NSPW2002)*, Virginia Beach, USA, Oct 2002.

[2] C. Bettini, S. Jajodia, X. S. Wang, and D. Wijesekera. Obligation monitoring in policy management. In *Proceedings of the IEEE Third International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*, Monterey, California, Jun 2002.

[3] C. Bettini, S.Jajodia, X. S. Wang, and D. Wijesekera. Provisions and obligations in policy management and security applications. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, Hong Kong, China, 2002.

[4] D. F. Brewer and M. J. Nash. The chinese wall security policy. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 206–214, Oakland, California, 1989.

[5] M. A. Brown. Conditional obligation and positive permission for agents in time. *Nordic Journal of Philosophical Logic*, 5(2):83–111, Dec 2000.

[6] D. Chiu, S. Cheung, and S. Till. A three layer architecture for e-contract enforcement in an e-service environment. In *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS-36)*, Hawaii, USA, 2002.

[7] F. Cuppens and C. Saurel. Specifying a security policy: a case study. In *Proceedings of the Ninth IEEE Computer Security Foundations Workshop*, Dromquinna Manor, Kenmare, County Kerry, Ireland, 1996.

[8] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *Proceedings of the 2nd International Workshop on Policies for Distributed Systems and Networks*, Bristol, UK, 2001.

[9] A. Daskalopulu, T. Dimitrakos, and T. Maibaum. E-contract fulfilment and agents'attitudes. In *Proceedings of the ERCIM WG E-Commerce Workshop on The Role of Trust in e-Business*, Zurich, Switzerland, 2001.

[10] N. Dulay, E. Lupu, M. Sloman, and N. Damianou. A policy deployment model for the ponder language. In *Proceedings of the IEEE/IFIP International Symposium on Integrated Network Management*, London, UK, 2001.

[11] B. S. Firozabadi and M. Sergot. Contractual access control. In *Proceedings of the 10th International Workshop of Security Protocols*, Cambridge, UK, Jun 2002.

[12] B. S. Firozabadi and M. Sergot. A framework for contractual resource sharing in coalitions. In *Proceedings of the IEEE Fifth International Workshop on Policies for Distributed Systems and Networks (POLICY'04)*, Yorktown Heights, New York, Jun 2004.

[13] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. In *Proceedings of the Intl. J. Supercomputer Applications*, 2001.

[14] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symp. Security and Privacy*, California, USA, 1982.

[15] Q. He and A. Antón. A framework for modeling privacy requirements in role engineering. In *Proceedings of the Ninth International Workshop on Requirements Engineering: Foundation for Software Quality, The 15th Conference on Advanced Information Systems Engineering (CAiSE'03)*, Klagenfurt/Velden, Austria, 2003.

[16] S. Jajodia, M. Kudo, and V. Subrahmanian. Provisional authorizations. *E-Commerce Security and Privacy*, 57(1):133–159, 2001.

[17] S. Kent, T. Maibaum, and W. Quirk. Formally specifying temporal constraints and error recovery. In *Proceedings of the RE93 - 1st Intl. IEEE Symp. on Requirements Engineering*, pages 208–215, San Diego, California, 1996.

[18] J. Lobo, R. Bhatia, and S. Naqvi. A policy description language. In *Proceedings of the National Conference of the American Association for Artificial Intelligence*, Florida, USA, 1999.

[19] J.-J. Meyer, R. Wieringa, and F. Dignum. *The Role of Deontic Logic in the Specification of Information Systems*, chapter 4. Kluwer Academic Publishers, 1998.

[20] Z. Milosevic, A. Josang, T. Dimitrakios, and M. Patton. Discretionary enforcement of electronic contracts. In *Proceedings of the 6th IEEE Conf. on Enterprise Distributed Object Computing (EDOC-2002)*, Lausanne, Switzerland, Sep 2002.

[21] N. Minsky and A. D. Lockman. Ensuring integrity by adding obligations to privileges. In *Proceedings of the 8th International Conference on Software Engineering*, pages 92–102, Klagenfurt/Velden, Austria, Aug 1985.

[22] N. Minsky and V. Ungureanu. Law-governed interaction: A coordination & control mechanism for heterogeneous distributed systems. *ACM Trans. Software Eng. and Methodology*, 9(3):273–305, July 2000.

[23] C. Molina-Jimenez, S. Shrivastava, E. Solaiman, and J. Warne. Contract representation for run-time monitoring and enforcement. In *Proceedings of the IEEE Conf. on Electronic Commerce (CEC)*, California, USA, June 2003.

[24] J. Park and R. Sandhu. The ucon abc usage control model. *ACM Transactions on Information and Systems Security*, Feb 2004.

[25] H. Prakken and M. Sergot. Contrary-to-duty obligations. *Studia Logica*, 57(1):91–115, 1996.

[26] C. N. Ribeiro, A. Zúquete, P. Ferreira, and P. Guedes. Enforcing obligation with security monitors. In *Proceedings of the Third International Conference on Information and Communication Security (ICICS'2001)*, Xi'an, China, 2001.

[27] C. N. Ribeiro, A. Zúquete, P. Ferreira, and P. Guedes. SPL: An access control language for security policies with complex constraints. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, California, Feb 2001.

[28] A. Schaad and J. Moffett. A framework for organisational control principles. In *Proceedings of the 18th Annual Computer Security Applications Conference*, Las Vegas, USA, 2002.

[29] F. B. Schneider. Enforceable security policies. *The ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[30] B. Shand and J. Bacon. Policies in accountable contracts. In *Proceedings of the IEEE 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*, page 80. IEEE Computer Society, 2002.

[31] G. Wasson and M. Humphrey. Policy and enforcement in virtual organizations. In *Proceedings of the 4th International Workshop on Grid Computing (Grid2003) (associated with Supercomputing 2003)*, Phoenix, USA, Nov 2003.

[32] G. Wasson and M. Humphrey. Toward explicit policy management for virtual organizations. In *Proceedings of the IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, Lake Como, Italy, 2003.

[33] T. Y. C. WOO and S. S. Lam. Authorizations in distributed systems: A new approach. *Journal of Computer Security*, 2(2,3):107–136, 1993.

[34] X. Zhang, J. Park, F. Parisi-Presicce, and R. Sandhu. A logical specification for usage control. In *Proceedings of the Symposium on Access Control Models and Technologies*, New York, USA, 2004.