A Scalable History-based Policy Engine

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

***Abstract*— The increasing complexity and heterogeneity in distributed systems is drawing system administrators into applying usage and access control policy engines. Higher-level policy languages allow policy administrators to demarcate themselves from implementation details, thus focusing on business rule definition. More specifically, history-based policies allow the specification of rules based on events that occurred in the past, such as separation-of-duty related rules (e.g. an employee cannot both issue a voucher and approve the payment).**

Several policy engines already support history-based semantics. However, they either provide limited expressiveness in policy rules or they neglect critical scalability issues. Individual policy definitions are disregarded in storage and lookup implementations, thus ignoring the potential for important performance optimizations. Furthermore, purging meta-policy semantics are not provided, inducing the growth of the past event repository until policy evaluation becomes unmanageable.

We present an extension to the Heimdall[1] system, a history-enabled policy engine which allows the definition, enforcement and auditing of history-based policies. This extension targets the scalability of Heimdall in practical environments, introducing an evaluation optimizer and the concept of purging meta-policy tags. An evaluation built on selected usage patterns corroborates the effectiveness of our approach, denoting encouraging performance results.

I. Introduction

Several usage 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[1]. These policies describe the access rules with an high level of abstraction, clearly separating the specification and the implementation of security mechanisms [2].

Furthermore, usage and access control is increasingly based upon advanced security patterns. The concept of history-based policies is particularly relevant, as it allows the inclusion of temporal events in the policy rules[3], [4], [5], [6]. Common examples include separation of duty policies[7], [8], which analyze previous actions in order to prevent a user from executing potentially conflicting operations[9] (e.g. issuing and paying the same check).

Several usage scenarios establish the relevance of history-based policies:

1) A financial department wants to assure that a person cannot both request an invoice payment and approve such payment.
2) In order to control costs, an organization defines monthly payment approval limits for its directors.
3) A doctor can only access medical data related to patients he assisted in the past.

Moreover, in a multitude of environments, such as those related to grid computing and peer-to-peer platforms, the high user and resource variability hinders the enforcement of usage and security policies employing only traditional access-control concepts. Effectively, models such as ACLs and RBAC lack the necessary expressiveness to apply a number of usage patterns, as the ones exemplified below:

1) A Grid Scheduler wants to assure a minimum CPU QoS to certain submitted jobs. Thus, it rejects job executions in nodes where CPU-intensive applications are already running.
2) A network-service provider wants to limit resource consumption to a certain amount per week (e.g. each user has a 10GB usage capacity each week). Additionally he wants to reserve 10% of the capacity for its local users.
3) A resource manager wants to restrict job submissions in situations where requests are continually rejected (e.g. due to resource exhaustion). Therefore, it automatically rejects any job from a user who had more than 2 job submissions rejected in the past 5 minutes.

Obviously, the usage patterns presented above can be enforced programmatically by developing custom extensions to the policy platform. However, such a solution not only introduces potential security vulnerabilities, but it also makes overall policy validation and maintenance difficult.

In addition, scalability concerns are seldom present in these ad-hoc policy implementations. This constitutes a serious problem in the case of history-based policies, because its enforcement generally involves lookups over a large set of past events (i.e. description of actions that occurred in the past). Thus, the operation of the policy platform in a real world environment quickly makes policy evaluation unmanageable, due to the continuous growth of the past event repository.

The concept of history-based policies has already been incorporated in several policy engines[10], [11], [12]. Although some performance optimizations were added to these platforms, none efficiently tackles the problems associated with the increasing size of the event repository.

We propose scalability optimizations to history-based policy engines based on two assumptions:

[1]Heimdall is the watchman of the gods in Norse Mythology. He guards Bifrost, the only entrance to Asgard, the realm of the gods.

- Most policies are based upon a specific subset of the past events (e.g. events containing a "Register" action). We introduce the concept of Custom Event Sets, that allow the defined event sets to be shared by several independent policies, thus reducing processing time. Additionally, we introduce an Event Set Optimizer, in order to establish the iterative update of the Custom Event Sets, in opposition to complete event set regeneration at evaluation time. As an important side effect, Custom Event Sets also contribute significantly to policy specification readability.
- Most events are only relevant for policy evaluation during a certain period in time (e.g. for a month) or until a certain condition is met (e.g. until the user has logged out). Thus, a critical addition to history-based engines is the concept of purging meta-policies. These policies allow irrelevant events to be safely removed from the past event repository (notice that events can be moved into a secondary repository, if necessary by legal and/or auditing reasons, rather than completely destroyed).

In summary, the platform presented in this paper, called Heimdall, allows the specification and enforcement of history-based policies. It proposes to tackle one of the most significant problems of history-based policy enforcement: scalability. The concept of Custom Event Sets, in conjunction with the Event Set Optimizer, allow an efficient management of subsets from the overall past events repository, thus increasing policy evaluation performance. Furthermore, Custom Event Sets contribute to policy specification readability and reuse. In parallel, we introduce purging meta-policies, that keep the past event repository in a manageable size, while assuring that only irrelevant events are removed. To our knowledge, this is the first practical design, implementation and evaluation of a platform that explicitly proposes a solution for history-based policy enforcement scalability.

This paper is organized as follows. In the next section we present a general overview of Heimdall, describing a generic application execution. In Section III we define several extensions to the xSPL policy language. We then present the architecture of the system in Section IV, describing the various components of the platform. In Section V, we focus on the prototype implementation, further evaluated in Section VI. Then, we examine Heimdall within this scientific field discussing some related work in Section VII. Finally, in Section VIII we present our conclusions.

II. System Overview

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

In order to allow an efficient specification and enforcement of history-based policies, we formulated the following design goals for Heimdall:

1) Provide an expressive, yet user-friendly, process for specifying history-based policies.
2) Implement a scalable and agile policy engine.
3) Implement a modular architecture that allows an easy integration with current runtime systems, while clearly separating policy specification and enforcement from application development.

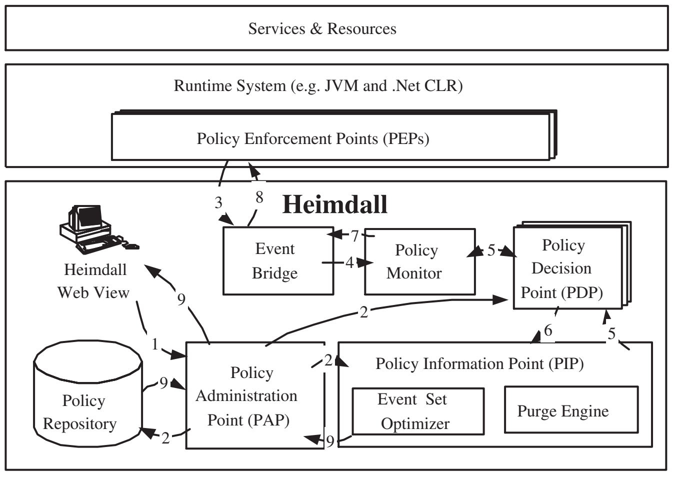
Services & Resources
Runtime System (e.g. JVM and .Net CLR)
Policy Enforcement Points (PEPs)
Heimdall
Heimdall Web View
Event Bridge
Policy Monitor
Policy Decision Point (PDP)
Policy Repository
Policy Administration Point (PAP)
Policy Information Point (PIP)
Event Set Optimizer
Purge Engine
1
2
3
4
5
6
7
8
9

Fig. 1. Overview of Heimdall.

The first goal is achieved by combining several factors. Heimdall's expressiveness is based upon an extension of SPL (Security Policy Language)[10], with significant extensions regarding policy scalability and past event repository management. On top of this language we developed a user-friendly web interface for policy specification and deployment. Heimdall Web View allows policy administrators to focus on business policy issues rather than on policy specification details. Furthermore, Heimdall Web View allows policy administrators to instantiate policy patterns (e.g. separation of duty, grid fair resource usage, etc) in an automatic and error-safe manner.

Concerning performance, Heimdall applies several optimizing and filtering techniques, such as automatically generated and iteratively updated subsets of past events that, in association with a purging mechanism (i.e. a mechanism that allows the removal of irrelevant events from the past event repository), significantly increase performance and scalability.

Finally, seamless integration with several runtime systems (e.g. the Java Virtual Machine, the .Net Common Language Runtime or the Globus Toolkit 4.0) is provided through Heimdall interface modules, and more specifically the Event Bridge (in the context of authorization policies). The input to Heimdall is generated in the runtime system Policy Enforcement Points by intercepting security critical operations (e.g. access to a webservice).

For clarity we illustrate in Figure 1 the access to a generic resource over a runtime system using Heimdall to enforce history-based policies.

1) The policy administrator defines the organization's policies using Heimdall Web View. Policy definitions (xSPL-based) are sent to the Policy Administration Point (PAP) for deployment and enforcement.
2) The PAP, after storing the policy definitions, generates a customized Policy Decision Point (PDP) for each one

of the deployed policies. In addition, it configures the respective purge definitions in the Policy Information Point (PIP).
3) A Policy Enforcement Point (PEP) in the runtime system sends an operation description to Heimdall.
4) 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 Policy Monitor for overall policy coordination.
5) The Policy Monitor contacts the Policy Decision Point (PDP) in order to determine if the current event is applicable for any of the deployed policies. The PDP retrieves applicable past events from the PIP in order to analyze the evolution of the policy.
6) If applicable for any of the policies, the current event is stored in the PIP for future evaluation. Furthermore, if the current event is specified in a Custom Event Set (c.f. Section III), the Event Set Optimizer is updated with relevant event information.
7) After overall evaluation is concluded, the Policy Monitor returns the authorization decision (i.e. an Authorize or Deny statement) concerning the execution of the specified action.
8) The Event Bridge translates the Authorization decision into the runtime system format and returns the security information to the PEP in the runtime system.
9) Periodically, the policy administrator can use the Heimdall Web View in order to analyze overall system status, checking aspects such as deployed policies, event repository size, purging definitions, among others.

Notice that Heimdall's expressiveness supports policy semantics beyond history-based policies. However, in the context of this paper we are specifically proposing scalability enhancements to history-based policies. We thus focus the architecture description on the modules related to this issue. For details regarding the enforcement of other type of policies in Heimdall see [13].

III. xSPL

Heimdall policies are defined in xSPL (eXtended Security Policy Language), an extension of the SPL language [10]. This language allows the definition of policies with complex constraints, including history-based policies.

A. xSPL Basic Constructs

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

$$< trigger\ expression :: decision\ expression >$$

The trigger expression specifies the applicability of the rule. If this expression is true then it means the decision expression part must be evaluated, and its result will constitute the authorization decision for the present rule.

One of the central concepts in xSPL is the event. Each action in the system, like a payment request or a resource consumption, is mapped into a normalized event construct. An event can either be instantiated in the Past Events Set (denoted as 'pe' in Figure 2), which represents the events previously executed, or it can represent the operation being evaluated, called the current event (always denoted as 'ce' in Heimdall implementation).

For clarity, we present in Figure 2 a definition in xSPL of a policy that restricts voting to users that have previously registered themselves in the system.

```
policy RegisteredVoting{
?RegisteredVoting():
  EXISTS pe IN PastEvents {
    ce.action = "Vote"
    ::
    pe.author = ce.author AND
    pe.action = "Register"
  }
}
```

Fig. 2. Example of a history-based policy in xSPL.

Although the task of defining this rule is several orders of magnitude easier than coding the rule into the authorization platform, we additionally allow policy administrators to instantiate policy patterns in Heimdall web interface (named Heimdall Web View).

B. Custom Event Set

Most history-based policy rules oblige the generation of a relevant subset of past events, obtained through the successive application of restriction conditions (e.g. all the login events for the current user which took place during last week) over the overall past event repository (existent in the PIP).

For clarity, consider the policy in Figure 3 that controls a payment process. It specifies that a user cannot both issue a payment order and subsequently authorize the payment himself. For simplicity and space reasons we'll use variations of this policy throughout the remainder of this paper.

```
policy PaymentApproval{
?PaymentApproval:
  NOT EXISTS pe IN PastEvents {
    ce.action = "Approve_Payment"
    ::
    ce.user = pe.user AND
    pe.action = "Pay_invoice" AND
    ce.action.invoiceID  = pe.action.invoiceID
}
```

Fig. 3. Example of a Payment Approval policy specified in xSPL without Custom Event Sets.

Representing the policy in such a way is definitely more user-friendly and maintainable than using a lower-level language directly (e.g. Java, .Net, C++, etc). However, the decision expression in the policy rule (i.e. the expression after the '::' sign), sequentially applies conditions to the global PastEvents set in order to determine relevant events (i.e. the "Pay Invoice" events belonging to the current user). This process must be performed every time the policy's trigger expression is true (i.e. whenever a "Approve Payment" action is requested). Furthermore, these event sets are independently specified and generated in other policies within the policy repository, even if identical.

This is obviously not efficient, thus in Heimdall we propose the specification of Custom Event Sets (illustrated in Figure 4 using the PaymentApproval policy). These sets not only improve the readability of the policy, but also and more importantly, allow the optimization of intermediary set generation results (e.g. the userActions event set could be used in other policies deployed in the system). This extension to the policy engine increases history-based policy evaluation performance, contributing significantly to the overall system scalability.

```
userActions = PastEvents @{.author = ce.user}
paymentOrders = userActions @{.action = "Pay␣invoice"}

policy PaymentApproval{
?PaymentApproval:
  ce.action = "Approve␣Payment"
  ::
  ce.action.invoiceID NOT IN paymentOrders
}
```

Fig. 4. Example of a Payment Approval policy specified in xSPL using Custom Event Sets.

In the PaymentApproval policy specification of Figure 4, we removed the existential operator (i.e. "NOT EXISTS pe IN PastEvents") and moved the conditions in the decision expression (i.e. 'ce.user = pe.user' and 'pe.action = "Pay invoice"') into two Custom Event Sets. The '@' operator basically restricts the set received as first argument using the conditions received as second argument (i.e. the condition '.action = "Pay invoice"' is applied to the members of the PastEvents set in order to generate the Custom Event Set)

The userActions Custom Event Set contains all the events that were executed by the current user, while the paymentOrders Custom Event Set further restricts this set to the user events that specify an invoice payment request. Notice that although the specified Custom Event Sets only possess a unique condition (e.g. .author = ce.user), their definition can potentially contain several conditions. We present in Figure 5 an alternative definition for the paymentOrders Custom Event Set:

```
paymentOrders = PastEvents @{.author = ce.user AND
       .action = "Pay␣invoice"
    }
```

Fig. 5. An Alternative Definition for the paymentOrders Custom Event Set.

For performance reasons, Heimdall indexes the Custom Event Sets by all the specified conditions (see more details in Section V).

These Custom Event Sets are managed by the Event Set Optimizer module, present in Heimdall Policy Information Point, and can be reused in any of the system policies.

In order to prevent unnecessary Custom Event Set generation, Heimdall (more specifically the Event Set Optimizer) associates these sets with dynamic update triggers (e.g. the paymentOrders Custom Event Set must be updated every time a "Pay Invoice" action is received), as described in Section IV.

C. Purging Past Events

Although the optimizing techniques described above significantly increase the policy engine's performance, a critical scalability problem remains. In a practical system, the event's repository size keeps increasing with system operation, until the sheer size of the repository makes policy queries unfeasible. The most common solution to performance deterioration is an ad-hoc repository cleanup, performed reactively by the policy administrator. We don't consider this a feasible approach, as it can raise inconsistencies in policy enforcement.

However, we notice that in most policy specifications, events are only relevant for policy evaluation during a certain period in time (e.g. for a month) or until a certain condition is met (e.g. until the user has logged out). We thus propose to remove events from the PIP in a policy-coherent manner, assuring they are no longer relevant for policy evaluation, and optionally moving them into a secondary repository if necessary for legal and/or audit reasons.

Hence, Heimdall adds several extensions to the xSPL language in order to allow policy administrators to define precise purging semantics. For instance, in the PaymentApproval policy specified above, both "Pay Invoice" and "Approve Payment" events could be removed after a payment has been approved, as such events are not necessary for further policy evaluations.

The PaymentApproval Purge definitions, represented in Figure 6, specifies that policy events should be purged every 7 days. Furthermore, it defines another Custom Event Set (userApprovedPayments) containing all the payment approvals for the current user. The Purge process should remove related operations (i.e. events) for which a matching payment request and approval exist in the event repository (there is a match if the action.invoiceID field is the same in both "Pay Invoice" and "Approve Payment" events). Notice that the Purge process operates in both the past events repository and the Event Set Optimizer' Custom Event Sets.

```
userApprovedPayments = userActions @{
       .action = "Approve␣Payment"}
PURGE–ID        : .approvals
PURGE–TRIGGER    : .action.invoiceID IN userApprovedPayments
PURGE–PERIOD     : 7 DAY;
```

Fig. 6. Purge Definitions for the PaymentApproval policy.

However, care must be taken in assuring the coherence of the overall policy set. Imagine that, in addition to the PaymentApproval policy defined above, the ApprovalLimit policy (represented in Figure 7) was added to the policy repository. It specifies that a user cannot approve payments summing more than 50,000 EURO each month.

The "Approve Payment" actions cannot be discarded from the PIP immediately after a payment approval, or otherwise

```
approvedPayments = PaymentApproval.userActions @{
        .action = "Approve_Payment"}
approvedPaymentsLast30Days = approvedPayments @{
        .time - time() < 30 DAY}

policy ApprovalLimit{
?ApprovalLimit:
  ce.action = "Approve_Payment"
  ::
  ce.action.value +
    approvedPaymentsLast30Days.SUM(.action.value)
    < 50000 EURO
}
```

Fig. 7. Example of a Payments Approval Limit policy in xSPL.

the ApprovalLimit policy is incorrectly enforced. Thus, at deployment time, Heimdall Web View informs the policy administrator about any purging conflict in the defined policies. With respect to the PaymentApproval and ApprovalLimit policies, a conflict notification is issued, as illustrated in Figure 8:

```
PURGE CONFLICT:
--- PaymentApproval Policy ---
--- PURGES ---
  userApprovedPayments = userActions @{
         .action = "Approve_Payment"}
  PURGE-TRIGGER   : .action.invoiceID IN userApprovedPayments
--- ApprovalLimit Policy ---
--- USES ---
  approvedPayments = PaymentApproval.userActions @{
         .action = "Approve_Payment"}
```

Fig. 8. Notification of a Purge Conflict in PaymentApproval and ApprovalLimit policies.

The notification points out that the PaymentApproval policy purge definition will potentially remove events that are relevant for ApprovalLimit's policy evaluation.

Heimdall will always take a conservative approach in what regards purge conflicts, thus assuring the coherence of policy enforcement, and overall system security. Thus, if the policy administrator disregards the purge conflict warning, Heimdall will associate a "CONFLICT" tag to the Purge definition, thus preventing its application. The "CONFLICT" tag is presented next to the policy definition in Heimdall Web View and allows the policy administrator to later conciliate the purge definitions, rather than completely neglecting the issue. In order to force the purge enforcement, the policy administrator must extend the PURGE-TRIGGER definition in order to remove the conflict. In the described scenario, the coherence could be maintained by specifying that 30 days must have passed before the purging:

```
PURGE-TRIGGER-EXTENDS : PaymentApproval.approvals;
PURGE-TRIGGER      : .time - time() > 30 DAYS;
```

Fig. 9. A PURGE-TRIGGER extension for the PaymentApproval policy.

Furthermore, as purging is critical for overall policy engine performance, Heimdall Web View allows a policy administrator to analyze possible scalability problems by presenting the events that are not being purged from the event repository (e.g. the "Pay Invoice" event in the PaymentApproval policy).

Finally, the Purge Engine also allows a policy administrator to purge unnecessary events without having to specify purge definitions for every policy in the system: the overall purge definition in Figure 10 is applied over the entire past event repository, and purges all those events which are more than 6 months old.

```
OVERALL-PURGE-TRIGGER     : .time - time() > 6 MONTHS;
```

Fig. 10. A Global Purge definition.

IV. Architecture

Heimdall possesses a modularized architecture based in several independent modules, depicted in Figure 1.

The Event Bridge receives action descriptions from the Policy Enforcement Points in the runtime systems, and creates a normalized description (referred to as an xSPL event). In a common Heimdall deployment, several Event Bridges can coexist, connected with several PEPs. These can denote different action descriptions and/or interfaces.

The Policy Decision Point (PDP) evaluates an event against policy definitions, returning an Allow, Deny or Not-applicable result.

The Policy Monitor is the overall policy coordinator in Heimdall. In the first place, it manages policy decision queries received from the PEPs (through the Event Bridge). These queries are forwarded to Heimdall Policy Decision Points (PDPs), which return an authorization decision.

The Policy Information Point (PIP), and associated Event Set Optimizer and Purge Engine, are the core of Heimdall history-based scalability mechanisms, which we focus in the context of this paper. They are described below in more detail.

The Policy Administration Point (PAP) interfaces with Heimdall Web View in order to provide the following functionalities to policy administrators:

- Manage (analyze, deploy, remove and modify) policies in Heimdall.
- Store policy definitions, along with policy patterns, in the policy repository.
- Obtain PIP statistics (number of events in the repository, scheduled purges, etc)

A. Policy Information Point (PIP)

The Policy Information Point is responsible for providing the PDPs all information that is relevant for policy evaluation. In traditional ACL and role-based policies, the PIP mainly interfaces with resources, subjects and the enclosing environment runtime system, in order to obtain pertinent information (e.g. current CPU load, number of users in the system, etc).

In the case of history-based policies, the PIP also has to store relevant past events, in order to allow policy evaluations.

Heimdall defines independent event repositories for each deployed policy, in order to reduce individual event respository size, and thus accelerate event queries. Although this approach demands additional storage, due to potential duplicated information, the performance gain justifies it. Notice that although each policy has its own event repository, it does not mean events are identical in each repository, as each policy can contain a distinct number of relevant events.

B. Event Set Optimizer

The Event Set Optimizer allows the evaluation of history-based policies without obliging the regeneration of each event subset at policy evaluation time. Recall the PaymentApproval policy in Figure 4. Whenever an ”Approve Payment” action is requested, the paymentOrders set must be traversed in order to verify if the current user also issued the corresponding payment order. Although this set can be generated during each evaluation, the process is rather time-consuming and, more importantly, the generation time increases with the number of events in the PIP, which constrains system scalability.

We propose to maintain the Custom Event Sets iteratively, updating them whenever relevant events are received. In particular, the PaymentApproval policy specifies two distinct Custom Event Sets, the userActions and the paymentOrders sets, containing respectively all the actions and the payment orders performed by the current user. Each of these Custom Event Sets is composed by a number of instances, depending on the set restriction conditions (e.g. the userActions set is composed by a distinct instance for each user and the paymentOrders restricts these instances, thus addressing only ”Pay Invoice” events). Although this approach generates additional storage expenditure, it significantly improves performance, which is paramount in policy engines when compared to ever-decreasing storage and memory costs.

In order to illustrate these concepts, we represent in Figure 11 a generic execution of a system enforcing the PaymentApproval policy defined in Figure 4 and the ApprovalLimits policy defined in Figure 7.

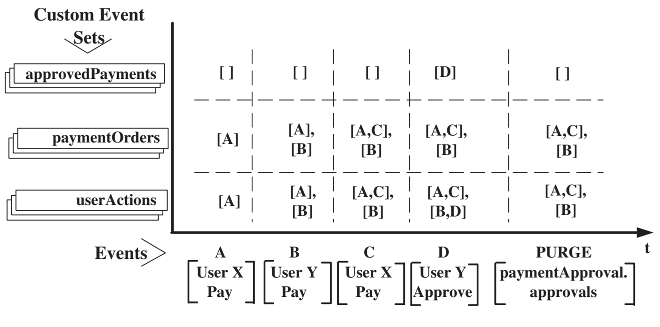

Fig. 11. The evolution of two history-based policies: PaymentApproval and ApprovalLimit

The Figure represents the content of the userActions, paymentOrders and approvedPayments Custom Event Sets while events are being processed by the policy engine. We assume that initially all sets are empty.

- User X and Y request several payment orders. The associated events (A, B and C) are inserted into the userActions and paymentOrders Custom Event Sets associated with each user.
- When User Y approves the payment order previously requested by user X (Event D), the Event Set Optimizer creates another entry for the approvedPayments Custom Event Set in order to introduce event D. As previously, this event is also inserted into the userActions Custom Event Set associated with user Y.
- Finally, the policy engine processes purging definitions associated with the ApprovalLimit policy. The event D, corresponding to a payment approval is withdrawn from the Event Set Optimizer in both userActions and paymentApprovals Custom Event Set instances (notice that the event is also removed from the global event repository).

Additional coherence issues must be overcome when the conditions specified in the Custom Event Sets restriction operator (@conditions) include time-based or set membership functions.

The first case is applicable when the events in the Custom Event Set are dependent upon the evaluation time. Consider the approvedPaymentsLast30Days Custom Event Set defined in Figure 7.

```
approvedPaymentsLast30Days = approvedPayments @{
        .time − time() < 30 DAY}
```

Fig. 12. A Custom Event Set dependent on the evaluation time.

This set should only contain the events which were generated during the 30 days prior to the policy evaluation. In order to maintain this semantics with the Event Set Optimizer mechanism, Heimdall automatically inserts an additional purge definition in the policy, stating that prior to every evaluation, the Custom Event Set should be purged of all the events that don’t respect the temporal restriction condition. This special kind of purge only affects the events in the Custom Event Set, disregarding the global event repository.

```
PURGE–ID          : .AUTOMATIC_TIME_01
PURGE–TRIGGER     : NOT (.time − time() < 30 DAY)
PURGE–PERIOD      : ON_EVALUATION
```

Fig. 13. An automatically generated Purge definition.

The second situation arises when Custom Event Set membership is dependent upon the existence of certain events in other Custom Event Sets. Consider for clarity the Custom Event Set defined in Figure 14:

The paymentsNotApproved Custom Event Set contains all the ”Pay Invoice” events for which a corresponding ”Approve Payment” event does not exist. The fact that the membership of an event in paymentsNotApproved is dependent upon the membership of other events in allPaymentApprovals also obliges a validation of the paymentsNotApproved Custom

```
allPaymentOrders = PastEvents @{
        .action = "Pay_Invoice"}
allPaymentApprovals = PastEvents @{
        .action = "Approve_Payment"}
paymentsNotApproved = allPaymentOrders @{
        .action.invoiceID NOT IN
        allPaymentApprovals}
```

Fig. 14. The paymentsNotApproved Custom Event Set is dependent upon the allPaymentApprovals Custom Event Set.

Event Set prior to policy evaluation. However, in this case, it is not sufficient to validate a PURGE-TRIGGER condition on all the events in the event set. The fact that the associated Custom Event Set (i.e. the allPaymentApprovals) can either have increased or decreased the number of events means that the trigger conditions would be excessively complex in order to accommodate all purging possibilities. Thus Heimdall rather associates a "MODIFIED" tag with the base event set (i.e. the allPaymentApprovals). When it needs to use the paymentsNotApproved event set, it verifies if the tag is active; in that case it regenerates all paymentsNotApproved instances.

C. Purge Engine

The Purge Engine assures the scalability of the history-based policy engine by constraining past event repository size. It periodically enforces the Purge definitions according to the semantics already presented in Section III.

The Purge Engine must perform lookups over the entirety of the event repository, but the fact that the periodicity of the purge process can be defined according to system processing capacity, keeps this process from significantly interfering with usual policy evaluations.

V. Implementation

Heimdall is implemented in Java, and deployed using Java Runtime Engine 1.4.2, in order to obtain a portable platform that can be deployed in a majority of environments.

The Event Bridge interfaces Heimdall with runtime system Policy Enforcement Points. It receives action requests and communicates the authorization decision. Currently we only support xSPL-based event descriptions, but a SAML[14]-enabled Event Bridge is being considered.

The Policy Monitor controls the overall deployment and enforcement of policies in the system. For that purpose, it holds a vector with the identifier of all deployed policies. Whenever an event is received from the Event Bridge, the Policy Monitor uses the Policy Decision Point to evaluate the event against all deployed policies.

The Policy Information Point 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 payment service, it does not 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 does not 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 [10]).

The Event Set Optimizer is generated dynamically for each policy, along with the Policy Decision Point. A repository for several event vectors is created for each of the Custom Event Sets specified in the policy xSPL definition (e.g. an event vector for each of the system users). The condition stated in the restriction operator creating the Custom Event Set is also used to define distribution rules for the events in each repository. For instance, the ".author = ce.user" condition indexes events in the vector according to the "ce.user" attribute. This allows the policy engine to search through the past events without having to lookup the entirety of the event repository. Events are inserted into the Event Set Optimizer, if relevant, immediately after they are authorized by the policy engine. They are only discarded from the optimizer instances when they become irrelevant for policy evaluation, by means of the Purge Engine or the Event Set Optimizer coherence mechanisms. The incorporation of Custom Event Sets in the policy definitions further increases the readability of the policy and fosters policy reuse.

The Purge Engine implementation is also generated dynamically according to policy purge definitions in xSPL. A different thread is associated with each policy, in order to activate the purge process at the defined periodicity.

VI. Evaluation

The extensions to Heimdall presented in this paper, and more specifically the Event Set Optimizer and the Purge Engine, aim to solve scalability problems currently faced by history-based policy engines. In order to assess the adequacy of Heimdall in what respects this goal, we performed a practical simulation in a Pentium 4, 2.8GHz, 512MB PC, running Windows XP Professional SP2.

Consider the PaymentApproval policy presented previously in Figure 4. We are assuming a scenario in which there are 250 active users in the system. Each one is inserting payment orders, which are later approved by a different user. Notice that in terms of policy evaluation performance, this is the worst possible scenario because, in order to validate the payment approval, the policy engine must search the entirety of the payment orders in order to confirm the user didn't previously issued a payment order with the same invoice number.

For space reasons, we restrict the presented evaluations to a single policy. However, further evaluations show that individual policy reevaluation time is independent of the number of policies in the system (e.g. evaluating an event in N similar policies takes about the same time as evaluating N events in an individual policy).

At simulation startup, each user keeps submitting payment orders in round-robin, until a certain number of payment cycles (i.e. a "Pay Invoice" event followed by a "Approve Payment" event) has been submitted to the policy engine (from 100,000 to 1,000,000 events). In this manner, we can simulate the load of the system after a large number of operations was executed. Afterwards, we perform additional batches of payment cycles, sustained until a certain time has elapsed. This allows us to obtain a reliable average for policy evaluation time for the pay/approve cycle.

We performed the simulation with various combinations in what respects the policy engine optimizations discussed in this paper, ranging from a fully optimized (Event Set Optimizer and Purge Engine) policy engine, to another policy engine lacking all optimizations. When the Purging Engine is active, purging is deployed every 5,000 payment cycles, and events are considered irrelevant after that same number of cycles. We obtained results for the event evaluation time, as discussed above, and also for the event repository dimension. The scale for the Y range in the chart of Figure 15 is logarithmic, because otherwise unoptimized results would not be perceptible.

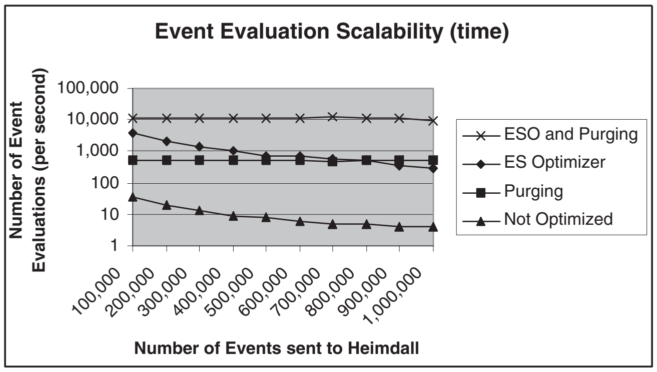

Fig. 15. System Evolution w.r.t. Pay/Approve Cycle Evaluation Time (notice that the Event Set Optimizer is denoted as ESO in the chart).

This chart shows that both the Event Set Optimizer and the Purge Engine represent a significant improvement to the policy engine w.r.t. performance, increasing the number of events evaluated each second by more than a order of magnitude.

As expected, when used in conjunction, they present the best results in this set of simulations. The Purge Engine by itself assures a stable event evaluation time; it surpasses the Event Set Optimizer optimization when performance gains due to the customized management of event subsets is minimized by the overall number of events in the log (at approximately 700,000 events).

In terms of event repository dimension, represented in Figure 16, the best results are obtained with the Purge Engine by itself. This is expectable, as irrelevant events are periodically removed from the event repository, and no significant data structures are needed for this optimization.

On the other extreme, we have the Event Set Optimizer optimization alone. Not only must the traditional event repository hold the entirety of the generated events (purging is not active), but in addition the events must also be stored in the Event Set Optimizer data structures. This fact doubles space requirements for the optimizer-enabled engine when compared

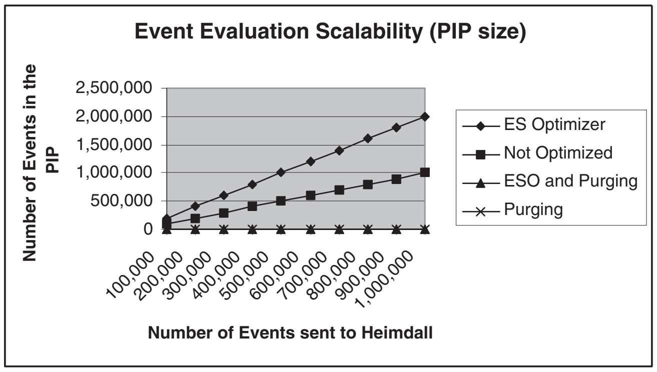

Fig. 16. System Evolution w.r.t past event repository Size (notice that the Event Set Optimizer is denoted as ESO in the chart).

to the engine lacking optimizations. Accordingly, the events that must be stored when optimizations are combined are double of those for the Purge Engine alone.

We conclude that the Purge Engine combined with the Event Set Optimizer is the best implementation alternative for the policy engine w.r.t. the scalability of the system. Performance is largely increased and event repository storage requirements are kept at a controlled size.

In sum, the evaluation shows that Heimdall scales efficiently with the number of events in the event repository, presenting itself as an adequate choice for policy enforcement in practical environments.

VII. Related Work

History-based policies have already been incorporated in several systems, and discussed in numerous papers.

Ribeiro [10] defines a security policy language, SPL, which includes the concept of history-based policies. He defines a global event repository where descriptions of all the executed actions are stored. Existential and universal operators are provided in order to select relevant events from this global repository, thus allowing an high level of expressiveness in policy rules. The internal organization of the event repository includes a filtering mechanism that selects only relevant event fields for storage. Furthermore, each policy is stored in independent hash tables, in order to increase performance. However, the hash table indexing function is only effective in a reduced number of situations (e.g. when there is only one restriction condition for an event subset, or all restrictions are specified in the policy's trigger expression), and its use can even be counterproductive (such as in the examples stated in this paper).

Dias[15] incorporated SPL in a mobile agents platform. The policies (including history-based policies) were used to conduct agent behavior in remote systems operation. In order to surpass significant performance problems, associated with moving the event repository from one remote system to another, Dias introduced the notion of a remote event repository for the Policy Decision Point. This approach however suffers from network latency and failure issues, hindering an efficient policy enforcement.

The LGI (Law-governed interaction) platform[16], [17] allows the enforcement of a system policy (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. The management (storage and retrieval) of events must be programmed into the policy definition itself, and the event repository (referred to as the Control State of the system) is stored as "a bag of Prolog-like terms". This hinders the implementation of complex policies, and further restricts storage optimizations. Heimdall, on the other hand, automatically generates the necessary Policy Decision Points (PDPs) from the policy definition. LGI's scalability analysis disregards the effects of event repository growth, which we consider a critical factor for policy engine performance.

The XACML (eXtended Access Control Modeling Language) [18] from the OASIS group, include the concept of history-based policies, as well as existential and universal operators. However, it lacks the concept of set restriction, which is one of the basis for Heimdall's Custom Event Sets. As a language specification, it does not oblige a specific implementation, and thus performance issues are not addressed.

Kent[19] discusses the specification of temporal constraints in policy rules by using deontic logic. Although this allows the formalization of the usage control model, no enforcement mechanisms were presented.

PDL (Policy Description Language)[12] allows the specification of past events in the policy rules. As in the case of Heimdall, Policy Decision Points are generated automatically at policy deployment time[20] and the events are stored in independent event history instances. However, policy evaluation is based upon the construction of non-deterministic finite automata, thus entailing serious scalability problems in the case of complex policies and/or large number of events in the repositories.

Park and Sandhu introduce the concept of pre-conditions in the context of a usage control model[21]. These conditions can be dependent on actions executed in the past, such as the number of times a certain operation was executed. This model is further refined by introducing several past temporal operators[22], in order to increase model expressiveness w.r.t. history-based policies. However, past events are not explicitly stored in an organized manner; thus, custom fields must be added by the policy administrator to the subject's attributes in order to allow this type of expressiveness (e.g. the policy administrator might need to define a user attribute containing the number of times that the user logged on during last week).

Deeds[23] is a history-based access control system. All policies must be written in Java, and event management must be custom developed for every policy (e.g. store a payment event or mark a login tag after user has logged in the system). Although this approach can foster efficient policy implementations, the ad-hoc development of PDPs and PEPs is impractical in large and/or complex systems. Heimdall, on the other hand, generates Policy Decision Points automatically, and events are managed in a transparent manner by Heimdall's Policy Information Point implementation.

Salsa[24] is a workflow-oriented mechanism. History-based functions were introduced on top of the original monitor server, in order to retrieve past events from the workflow log, and identify event dependencies. The system lacks a scalability evaluation w.r.t. history-based policies, as well as details on the policy language expressiveness.

Adage[11], [25] is a generic authorization policy engine for distributed systems. It introduces the concept of history-based separation of duty. This model assumes that a user can be allowed to assume conflicting roles at the same time, as long as the executed actions don't constitute a conflict themselves (a user can simultaneously issue and pay a check, as long as it isn't the same check). This level of expressiveness is also supported by Heimdall. However, Adage does not include the concepts of Custom Events Sets nor of Event Purging, thus lacking scalability mechanisms.

Sandhu[26] proposes a mechanism for separation of duty enforcement. Due to the specialized nature of policy enforcement, Sandhu refers that after operation completion (e.g. issuing and payment of a voucher) the associated events can be discarded. Heimdall supports this approach for separation of duty policies and further allows customized purging of the event repository for other kinds of policies.

Stream[27] is a Data Stream Management System (DSMS) that could potentially be used to enhance Heimdall Policy Monitor processing capabilities. It provides a language named CQL (Continuous Query Language)[28] that allows the filtering of relevant records in the data stream, thus enabling a number of performance optimizations. However, Heimdall semantics often oblige the processing of the entire history repository, hindering traditional stream-based processing.

We thus conclude that all the analyzed systems lack the necessary expressiveness to specify purging semantics. In addition, they generally disregard event set optimizations which are implemented by Heimdall, such as the concept of Custom Event Sets.

VIII. Conclusions

In this paper we evidenced the relevance of history-based policies. Policy engines that lack such expressiveness oblige policy administrators to code usage and access rules into the policy platform itself, thus introducing potential security vulnerabilities. We further pinpointed several scalability problems in the implementation of history-based policy rules.

We introduced the concept of Custom Events Sets, constituted by subsets of the past event repository, relevant for a specific policy evaluation. The Event Set Optimizer manages Custom Events Sets in order to prevent unnecessary set regeneration. This approach represents significant performance improvements, together with improved policy readability.

Furthermore, Purging Meta-Policy tags allow irrelevant events to be removed from the system in a controlled and coherent manner.

These solutions were implemented in the Heimdall Policy Engine. The evaluation of selected usage scenarios show that the system can cope with a large number of events in the Policy Information Point and continuous intensive operation, thus denoting encouraging results.

References

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

[2] T. Y. C. WOO and S. S. Lam, "Authorizations in distributed systems: A new approach," *Journal of Computer Security*, vol. 2, no. 2,3, pp. 107–136, 1993.

[3] C. de Laat, G. Gross, L. Gommans, J. Vollbrecht, and D. Spence, "Rfc 2903 - generic AAA architecture," IETF, Tech. Rep., August 2000.

[4] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian, "Flexible support for multiple access control policies," *ACM Transactions on Database Systems (TODS)*, vol. 26, no. 2, pp. 214–260, 2001.

[5] 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.

[6] R. S. Sandhu and P. Samarati, "Access control: Principles and practice," *IEEE Communications Magazine*, vol. 32, no. 9, pp. 40–48, 1994.

[7] Gligor, Gavrila, and Ferraiolo, "On the formal definition of separation-of-duty policies and their composition," in *Proceedings of the 19th IEEE Computer Society Symposium on Researh in Security and Privacy*, 1998.

[8] M. J. Nash and K. R. Poland, "Some conundrums concerning separation of duty," in *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, Oakland, USA, May 1990.

[9] N. Minsky and V. Ungureanu, "Unified support for heterogenous security policies in distributed systems," in *Proceedings of the 7th USENIX Security Symposium*, Texas, USA., January 1998.

[10] 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.

[11] R. T. Simon and M. E. Zurko, "Separation of duty in role-based environments," in *Proceedings of the IEEE Computer Security Foundations Workshop*, 1997.

[12] 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.

[13] P. Gama and P. Ferreira, "Obligation policies: An enforcement platform," in *Proceedings of the Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'05)*, Stockholm, Sweden, June 2005.

[14] S. Cantor, J. Kemp, R. Philpott, and E. Maler, "Assertions and protocols for the oasis security assertion markup language (saml) v2.0," OASIS, Tech. Rep., March 2005.

[15] P. Dias, C. Ribeiro, and P. Ferreira, "Enforcing history-based security policies in mobile agent systems," in *Proceedings of the IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, Lake Como, Italy, 2003.

[16] N. Minsky and V. Ungureanu, "Law-governed interaction: A coordination & control mechanism for heterogeneous distributed systems," *ACM Trans. Software Eng. and Methodology*, vol. 9, no. 3, pp. 273–305, July 2000.

[17] N. Minsky, Y. M. Minsky, and V. Ungureanu, "Making tuple spaces safe for heterogeneous distributed systems," in *Proceedings of the 2000 ACM Symposium on Applied Computing*, Como, Italy, 2000, pp. 218–226.

[18] T. Moses, "extensible access control markup language (xacml) version 2.0," OASIS, Tech. Rep., February 2005.

[19] S. Kent, T. Maibaum, and W. Quirk, "Formally specifying temporal constraints and error recovery," in *Proceedings of the RE'93 - 1st Intl. IEEE Symp. on Requirements Engineering*, San Diego, California, 1996, pp. 208–215.

[20] J. Chomicki and J. Lobo, "Monitors for history-based policies," in *Proceedings of the Second IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'01)*, London, UK, June 2001.

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

[22] 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.

[23] G. Edjlali, A. Acharya, and V. Chaudhary, "History-based access control for mobile code," in *Proceedings of the 5th ACM Conference on Computer and Communications Security (CCS-5)*, San Francisco, USA., November 1998.

[24] M. H. Kang, J. S. Park, and J. N. Froscher, "Access control mechanisms for inter-organizational workflow," in *Proceedings of the sixth ACM Symposium on Access Control Models and Technologies*, Chantilly, Virginia, USA, May 2001.

[25] M. E. Zurko, R. Simon, and T. Sanfilippo, "A user-centered, modular authorization service built on an rbac foundation," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, Oakland, USA., 1999.

[26] R. S. Sandhu, "Separation of duties in computerized information systems," in *Proceedings of the IFIP Workshop on Database Security*, 1990.

[27] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, *STREAM: The Stanford Data Stream Management System*, 2004.

[28] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: Semantic foundations and query execution," *To appear in VLDB Journal*, 2005.