# An experimental testbed to predict the performance of XACML Policy Decision Points

Bernard Butler, Brendan Jennings, and Dmitri Botvich
FAME, Telecommunications Software & Systems Group (TSSG)
Waterford Institute of Technology
Waterford, Ireland
Email: {bbutler, bjennings, dbotvich}@tssg.org

***Abstract*—The performance and scalability of access control systems is a growing concern as organisations deploy ever more complex communications and content management systems. This paper describes how an (offline) experimental testbed may be used to address performance concerns. To begin, timing measurements are collected from a server component incorporating the Policy Decision Point (PDP) under test, using representative policies and corresponding requests. Our experiments with two XACML PDP implementations show that measured request service times are typically clustered by request type; thus an algorithm for request cluster identification is presented. Cluster characterisations are used as inputs to a PDP performance model for a given policy/request mix and an analytic (queueing) model is used to estimate the equilibrium server load for different mixes of request clusters. The analytic *performance prediction* model is validated and extended by discrete event simulation of a PDP subject to additional load. These predictive models enable network administrators to explore the capacity of the PDP for different overall loadings (requests per unit time) and profiles (relative frequencies) of requests.**

## I. Introduction

Access control systems apply policies to ensure that *Subjects* can access *Resources* if and only if they are entitled to do so. In the standard architecture [1], access requests are sent to Policy Execution Points (PEPs), which hand off the access decision itself to a Policy Decision Point (PDP). The PEP is largely stateless and so scales outwards easily. However, the PDP needs to consult a policy set and apply the rules therein to *Permit* or *Deny* each request and so can become a performance bottleneck. Thus PDP performance is an important characteristic of access control requirements in deployed ICT systems. This observation is especially true in large and complex organisations, where access decisions depend on the (rich) context of the access request. *Fine-grained* access control enables system administrators to implement security policies with complex decision boundaries but also leads, in general, to more complex policy sets, resulting in longer PDP search times. Fine-grained access control also requires the PDP to check more access requests within a session. As an example, in a single session, Subjects may wish to exchange Resources with media type 1 (voice or plain email) and media type 2 (email or IM file attachments). Notably, Chinese-Wall fine-grained access control policies suffer scalability problems; Cisco colleagues see this in real world deployments. For the scenarios in our paper, policy characteristics are captured implicitly in service time measurements. As with any security deployment, it is necessary to respond rapidly as new threats arise, so *dynamic* updates to policies are necessary. This need to support policy sets that evolve over time makes it more difficult to use caching and similar strategies to improve PDP performance and scalability.

By instrumenting the SunXACML open source PDP implementation, we noticed that the execution time of many of the steps taken by the PDP do not depend on its data (i.e., the policy set and incoming requests). The major exception to this observation is the step where the PDP seeks to match the request against the policy set. This is a complex search problem, depending on many factors such as the number of rules sharing a Target that matches the request, how deeply nested the policy set is and what rule- and/or policy-combining algorithm is in force. Rather than trying to build a (fragile) explicit model for service times, we collect timing observations and build a simpler model based on request clusters. The model is sufficiently simple to calculate some properties analytically; for other properties, we use simulation. Summarising, this experimental approach operates at the level of request *ensembles* rather than individual requests. While we lose some detailed insight, we gain a flexible model that can be updated easily (e.g., in respect of clusters of observed requests).

The testbed can be used for two purposes:

- *Comparison*
  To estimate the effects of an experimental treatment under controlled experimental conditions, by comparing cases with or without that treatment. Treatments might include projected PDP improvements, increased policy set size, etc.
- *Prediction*
  To estimate a performance metric given a new set of conditions, e.g., a change in the access request mix, or rapid changes in request arrival rates.

As seen in Section VII, our analytic and simulation models can be used for the important objective of PDP performance prediction. Practical applications include PDP dimensioning (either pre-deployment or mid-deployment when conditions change). Resolving PDP performance issues is out of scope for this paper.

These uses are explored in Sections V and VI, and Section VII describes some initial experiments covering both.

When used in *prediction* mode, the explicit analytic model and simulation approaches are complementary and are presented in Section V . The analytical model is more convenient but simulation can be used in more scenarios. Section VII also indicates how each can validate the other, by choosing a scenario in which both approaches can be applied to the same data and are shown to produce equivalent results. In particular, simulation enables powerful analysis of "what if?" scenarios relating to expected changes in policy sets and/or system user behaviour—as reflected in changing policy request types and arrival patterns. Crucially, the simulation experiments are grounded in actual measurements from real PDPs, thereby reducing threats to their *construct validity* [2]. Because of the extensive instrumentation in the testbed, collecting service time measurements is much easier than in a "production" deployment so it can be done more often. Thus we can also monitor the performance impact of policy changes, by analogy with the use of Margrave to do (logical) policy change impact analysis [3].

## II. Related Work

There is extensive literature both on policy authoring [4] and on policy testing [5]. Much of the focus has been on ensuring that the policy set is maintainable, correct, comprehensive and consistent [6]. If an access control *system* is to be "fit for purpose", we contend that such requirements are necessary but not sufficient. One requirement missing from that list is usability for end users, in which system performance plays a big rôle.

Measurement-based simulation for performance modelling and enhancement has a long history. Sometimes it offers the only practical approach for modelling the behaviour of a complex system under extreme conditions. In Section I, we identified the XACML policy search step as being the most difficult to model. In recent years, researchers have turned their attention to improving the performance of policy evaluation in general and that of XACML-encoded policies in particular [7]. Promising techniques include policy reconfiguration [8], recoding [9], query rewriting [10], [11] and policy simplification using Description Logics [12]. Anecdotally, policy and rule set size and complexity cause some problems but we have not investigated this claim directly using our testbed yet. For a given performance improvement technique, it is difficult to predict whether the technique brings *material* benefits in PDP performance. [13] study (pdp x policy x request) comparisons using requests serviced per second as a metric, a concern for *clients*. By contrast, our policy set is fixed but we vary many other factors and use server utilisation as the metric, since this is a concern for *server* dimensioning. We also derive an analytical model and perform *measurement-based simulation*. This paper builds upon earlier work [14], which describes *comparison* experiments using an earlier version of the testbed. The additional contributions in our present paper are (1) an improved clustering algorithm and (2) analytical and simulation models based on service time measurements and identified clusters, with a new focus on *prediction* of server utilisation and hence dimensioning.

## III. Framework Overview

In this paper, we study PDP performance by considering its sensitivity to factors such as request mix, request arrival rate and PDP implementation. To achieve this aim, we create a testbed in which each of these factors (and others) can be controlled independently. We consider access control policies with the only restriction being that the policies can be specified in XACML 2.0 (an OASIS standard). The main components of the testbed are shown in Figure 1, notably

- The *client* (`xtc`) produces requests that are subsequently relayed to the server component (`xts`). `xtc` can obtain its requests from several sources, such as: resampled from an existing request set (MODE 1 in Figure 1); generated through analysis of the deployed policy set (MODE 2 in Figure 1); or via a domain model of the behaviour of users of the system within which the server is deployed (MODE 3 in Figure 1).
- The *server* (`xtc`) comprises a PDP implementation and an adapter to handle messages coming from the PEP in the `xtc` client. Therefore, as seen by the client, the PDP is a black box with a standard, simplified API. The main functionality exposed through that API is 1) read a policy set and 2) evaluate an access request. The adapter is responsible for measuring and recording the service time per request. Usually PEPs act as intermediaries between clients and PDPs, but in its present form, the testbed PEP is very simple.
- The *analyser* component (`xta`) aggregates and enhances the raw timing measurements, to provide accurate service time measurements and derived quantities (such as cluster assignments) with statistical analysis for comparison experiments.
- The *predictor* (`xtp`) is where the explicit analytic model is implemented together with the discrete event simulator for more complex scenarios.

Provided the policies and requests are syntactically valid, the framework applies to any XACML policy set. Moreover, since we treat the PDP as a black box, it can be extended to other PDP types, or even to general request/response systems.

## IV. Data preparation

The timings obtained via the Adapter capture the total time spent by the PDP per request a) converting the XACML-encoded request into the PDP's internal representation in memory, b) searching the policy set for matching policies and c) returning the decision as a XACML-encoded response. A model based on individual requests would be too fine-grained and difficult to generalise. However, PDP service times appear to exhibit clustering behaviour, i.e., their distribution is a mixture of simpler distributions. The algorithm used to process the raw service times to provide *clustered* measurement data for simulation purposes is presented below.

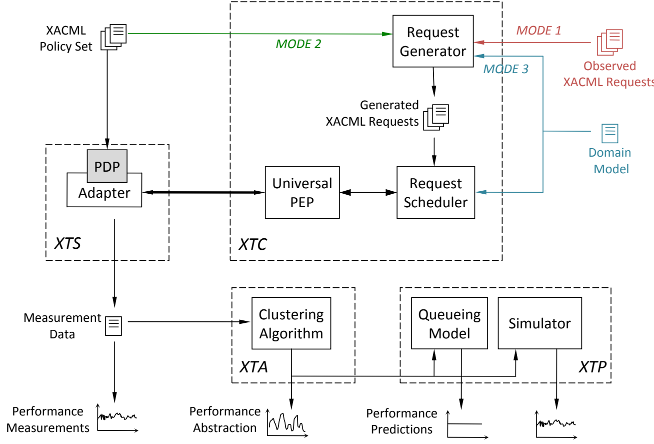



Fig. 1. The main measurement system and simulator components.

Let $t = t(S, P; R, q) \in \mathcal{R}^{u \times q}$ be the set of PDP service times, where $S$ represents (characteristics of) the PDP server, $P$ represents the policy set to search, $R$ is the set of requests, $r = |R|$, $U$ is the combination of $S \times P \times R$ experimental conditions, $u = |U|$ and $q$ is the number of replicate measurements of $t$, holding conditions $S, P, R$ fixed.

Algorithm Step 1 removes anomalous service times by choosing the minimum of the replicate service times for each $S \times P \times R$ combination of experimental conditions. Step 2 computes the (probability) density of service times for each $S \times P$ combination, based on the $r$ available service times for that combination. Step 3 inspects the service time density function for each $S \times P$ combination and estimates the number $n$ of request clusters. Step 4 computes a function of each service time distribution such that the minima of this function are candidate cluster centres. Step 5 labels requests according to their membership of the service time clusters, for each of the $S \times P$ service time distributions. Step 6 estimates the mean and variance of the Gaussian distribution fitted to service times of requests in the $|S| \times |P| \times n$ clusters. Apart from user intervention in Step 3, it is fully automatic. A more formal statement of the algorithm can be found in Figure 2.

The mean and variance of each derived cluster can then be used to simulate the service times of large numbers of requests belonging to the cluster. This algorithm was implemented in R [15].

## V. Analytic and Simulation Models

We recall that the PDP receives requests, consults a policy set and emits responses. We note from Figure 4 that service times do not follow a simple distribution, so estimates of the mean processing time need to take account of request frequencies. Requests are generated by a stochastic process, so queueing will occur except in (uninteresting) cases where request interarrival times are much greater than request service times.

1: **for** $i$=1 to $|S|$ **do**
2: **for** $j$=1 to $|P|$ **do**
3: Let $s_i$ and $p_j$ be the server and PDP instance, respectively. Hereafter indexes $s_i$ and $p_j$ are implicit.
4: Let $\mathbf{t} = \mathbf{t}(r_k)$ be the vector of replicate service times for the request indexed by $r_k$.
5: Compute $\underline{t}(r_k) = \min(\mathbf{t}(r_k))$ as the service time obtained by selecting the minimum of the replicate service times for a particular request $r_k$
6: Compute $d(\underline{t})$ as the density function of service times.
7: By inspection of the density plot $d(\underline{t})$, choose the number $n$ of significant density peaks, equivalent to $n$ the number of request clusters - see Figure 4.
8: Compute $\tilde{f}(d) = f(t) = d(t)\ddot{d}(t)$ at *discrete* $t$ values. The minima of $f(t)$ are indicative cluster centres.
9: Sort $f(t)$, select the first $\hat{n} = n-1$ values and lookup the corresponding centres.
10: Compute the inner cluster endpoints $\{t_c^{(p)}, p = 1, \ldots, \hat{n}\}$ by linear interpolation.
11: Assign $n$ cluster intervals $[t_c^{(p)}, t_c^{(p+1)}]; p = 0, \ldots, \hat{n}$ where $t_c^{(0)} = 0$ and $t_c^{(n)} = \infty$.
12: Label each request $r_k$ with its cluster index $p$ based on the interval into which its service time $\underline{t}(r_k)$ fits.
13: Fit a Gaussian to service times in each cluster $p$.
14: **end for**
15: **end for**

Fig. 2. Algorithm to cluster service times

### A. Mean Value Analysis

A PDP can be modelled as a queue: requests arrive with mean arrival rate $\lambda$ and exit with rate $\mu$. For the queue length to be bounded, we require $\rho < 1$, where $\rho = \lambda\bar{x}$, where $\bar{x}$ is the mean service time. In typical deployments, the arrival process may be nonstationary, e.g., request arrival rates are greater during working hours. However, in the simplest case, the arrival process is memoryless and hence the arrival times have an exponential distribution, as assumed in this paper. We consider nonstationary entensions to the model in Section V-C. However, the simplest queueing model is $M/M/1$ with FIFO scheduling. Since the measured service times are known to be clustered, the queue does not satisfy the assumptions of the $M/M/1$ model. Instead, we model the queue as $M/G/1$, i.e., access requests are generated by a Markov process (hence arrivals are memoryless), but the service times are drawn from a "General" distribution. Because of the presence of request clustering, we choose to model PDP service times as being drawn from a hyperexponential distribution (essentially, a weighted sum of exponentials); see Equation 5. Each exponential term takes its parameters from a measured request cluster. The weights combining the exponential distributions depend on the arrival rates of the different request clusters.

The PDP utilisation, (equivalently: mean load on the server

at equilibrium) is

$$\rho = \lambda\bar{x}, \text{ where } \rho < 1 \text{ for the queue to remain bounded} \quad (1)$$

By definition, the coefficient of variation $C_b$ of the service time distribution with density function $b(x)$ is defined by

$$C_b^2 \stackrel{\text{def}}{=} \frac{\sigma_b^2}{\bar{x}^2} \quad (2)$$

where

$$\begin{aligned} \bar{x} &\equiv E\{X\} = \int_0^\infty x b(x)\, dx \\ \sigma_b^2 &\equiv E\{X^2\} - (E\{X\})^2 = \int_0^\infty x^2 b(x)\, dx - \bar{x}^2. \end{aligned} \quad (3)$$

The Pollaczek-Khinchin mean-value formula for queue length at departure instants [16, Eq 5.63]

$$\bar{q} = \rho + \rho^2 \frac{(1 + C_b^2)}{2(1-\rho)}, \quad (4)$$

is an explicit formula in terms of the quantities defined in Equations 1 and 2.

For hyperexponentially-distributed service times, the service density function is

$$b(x) \stackrel{\text{def}}{=} \sum_{i=1}^{p} \alpha_i\, \mu_i e^{-\mu_i x}, \text{ where } \sum_{i=1}^{p} \alpha_i \equiv 1 \equiv \int_0^\infty b(x) dx \quad (5)$$

Substituting Equation 5 into Equation 3 gives

$$\begin{aligned} \bar{x} &= \sum_{i=1}^{p} \frac{\alpha_i}{\mu_i} \\ \sigma_b^2 &= \sum_{i=1}^{p} \frac{2\alpha_i}{\mu_i^2}. \end{aligned} \quad (6)$$

Note that $\mu_i$ and $\frac{1}{\mu_i}$ are the mean service rate and mean service time, respectively for cluster $i$. We can substitute Equation 6 in Equation 2 and hence in Equation 4 to obtain $\bar{q}$.

Therefore, given $p$ request clusters, with measurements of the mean service time per request cluster $\mu_i$, we can compute expected queue lengths $\bar{q}$ for different request cluster mixes $\alpha_i, i = 1, 2, \ldots, p$.

We can also compute the mean queue waiting time using [16, 5.70]

$$W = \rho \frac{(1 + C_b^2)}{2(1-\rho)} \bar{x} \quad (7)$$

### *B. Service times and arrival rates*

We note that the arrival rate of each cluster-serving component is the product $\alpha_i \lambda$ of

- the relative frequency of requests belonging to that cluster: $\alpha_i$
- the global arrival rate, ignoring cluster membership: $\lambda$

Because of the way $\alpha_i$ is defined,

$$\lambda \equiv \sum_{i=1}^{k} \alpha_i \lambda = \sum_{i=1}^{k} \lambda_i. \quad (8)$$

The user needs to specify the (per-cluster) mean interarrival times $\frac{1}{\lambda_i}$ and the measurement-derived mean service times $\frac{1}{\bar{x}_i}$ of the discrete event simulation. The mean service time is estimated by computing the weighted mean of the individual cluster service means. In practice, $\alpha_i$ would be found by characterising and hence *calibrating* actual access request traffic.

Using the cluster assignments $C(r_i) = j$ ($r_i$ being the $i^{\text{th}}$ request type and $C$ being the function mapping $r_i$ into cluster index $j$) from the measurements above, we can

1) Compute the mean service time $\bar{x}$ using Equation 6
2) Estimate the capacity (the maximum arrival rate $\lambda$ such that the queue length remains acceptable ($\rho < R$ where $R < 1$) of the PDP server used to generate the measurement data above for a *given mean service time*.

### *C. Extending the model: steady state plus overload*

Because the request arrivals (both baseline and overload) are generated by a (memoryless) Markov process, overload requests can be modelled separately from baseline requests. That is,

$$\rho = \rho^{(\text{base})} + \rho^{(\text{overload})} \quad (9)$$

where, in general terms, the utilisation

$$\rho^{(\odot)} = \lambda^{(\odot)} \bar{x}^{(\odot)} \quad (10)$$

and the general service mean

$$\bar{x}^{(\odot)} = \sum_{j=1}^{n} \alpha_j^{(\odot)} \bar{x}_j \quad (11)$$

Let $\lambda^{(\text{overload})} = \gamma \lambda^{(\text{base})}$ where $\gamma$ is the overload factor; then

$$\begin{aligned} \bar{x}^{(\text{base})} &= \sum_{j=1}^{n} \alpha_j^{(\text{base})} \bar{x}_j \\ \bar{x}^{(\text{overload})} &= \sum_{j=1}^{n} \alpha_j^{(\text{overload})} \bar{x}_j \end{aligned} \quad (12)$$

So

$$\begin{aligned} \rho^{(\text{base})} &= \lambda^{(\text{base})} \sum_{j=1}^{n} \alpha^{(\text{base})} \bar{x}_j \text{ as before;} \\ \rho^{(\text{overload})} &= \gamma \lambda^{(\text{base})} \sum_{j=1}^{n} \alpha_j^{(\text{overload})} \bar{x}_j \end{aligned} \quad (13)$$

The base arrival rate can be computed from the base utilisation and base service times:

$$\lambda^{(\text{base})} = \frac{\rho^{(\text{base})}}{\sum_{j=1}^{n} \alpha^{(\text{base})} \bar{x}_j} \quad (14)$$

Note that the free parameters in Equation 13 are $\gamma$ and $\{\alpha_j^{(\text{overload})}, j = 1, \ldots, n\}$; all other parameters are either measured directly or computable from measurements.

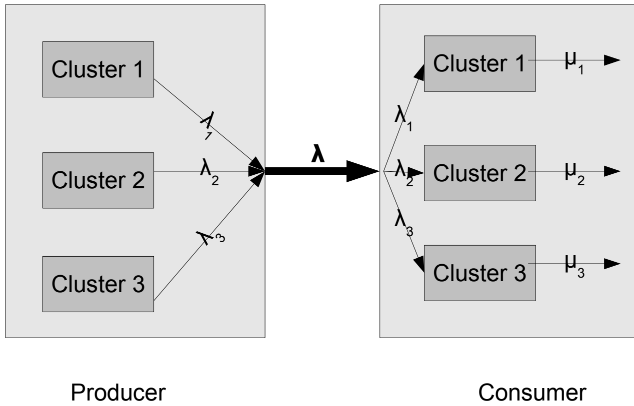



Fig. 3. Decomposing the simulation request token producers and consumers into cluster-specific components.

### D. Simulation

While the explicit analytical model is attractive and convenient for sensitivity analysis and other uses, it is not sufficient:

- explicit formulae are unknown for quantities such as the queue length variance
- known formulae evaluate mean values only, reflecting long-term queue evolution not transient effects
- if an explicit service time distribution model is not available, explicit formulae will not exist.

To overcome these limitations, we developed a simulation model. Following the explicit model, we model the XACML PDP as a single processor, serving a single queue and employing a FIFO (First-In-First-Out) queueing discipline. Arriving XACML requests are placed at the tail of the queue and served in order of arrival. For the M/$H_k$/1 queue corresponding to our analytic model, we view the simulated PDP as comprising $k$ disjoint components, each associated with a single request cluster. Request tokens are produced by $k$ Markov processes representing the $k$ clusters of requests. The token generation rate of each Markov process becomes the interarrival rate of the queue for the appropriate PDP cluster-specific component. By this device, we decouple request token generation and consumption into separate per-cluster streams; see Figure 3.

## VI. Framework Deployment

The policy set used in all trials described in this paper is the 'continue-a' set referenced in [3] and obtained as part of the Xengine PDP source distribution. The policies govern submission of papers to a notional conference. The policy set was loaded into each PDP policy repository. Four hundred representative requests (two hundred from each of the 'single' and 'multi22' request sets from [3]) were issued against the server and the timings were recorded in a text file. This process was repeated 100 times (with the order of the requests being randomised in each replication) on a server instance (hosting the `xts` component) that was otherwise idle, to minimise the effect of anomalous timings (if a background process started, say).

| host | pdp | Request Group |
|---|---|---|
| bear | SunXACML | single<br>multi22 |
| | Enterprise XACML | single<br>multi22 |
| inisherk | SunXACML | single<br>multi22 |
| | Enterprise XACML | single<br>multi22 |

TABLE I
Main experimental conditions for the trials

Balanced full factorial trials were run as indicated in Table I.

Each host runs a recently patched Ubuntu 10.04 operating system. They have identical versions of applications such as Java, R etc. The same testbed *source* code is deployed on each. Both use dual-core 64-bit Intel processors. They differ in that 'bear' has a 32-bit operating system rather than a 64-bit operating system as on 'inisherk'. They also have different motherboards and memory configuration. 'inisherk' is about two years newer than 'bear' and hence might be expected to have generally lower service times, however we cannot assume that all requests will be subject to the same speedup factor and hence that cluster membership will be identical irrespective of the host.

The two PDP implementations are representative of different design goals. SunXACML was designed as a reference implementation, Enterprise XACML as a more practical implementation, with its developers specifically claiming good performance [17]. They were developed independently and hence might be expected to exhibit different service time clustering behaviour.

The XACML structural differences between the 'single' and 'multi22' *request groups* are not the focus of this paper, rather the fact that their service times might be expected to cluster differently.

The 16 cases in Table I summarise a more detailed experiment in which there are 100 replicate measurements on each of the 200 request types in the specified request group.

Each arrival weight $\alpha_j$ depends on the arrival rate of requests in cluster $j$ relative to requests from all clusters. Ideally $\alpha_j$ would be computed by observing the frequency of requests in an *actual* deployment. For the purpose of this scenario, we assume request types have identical arrival rates, in which case $\alpha_j$ is the relative size of cluster $j$.

The simulation model uses the OPNET$^{TM}$simulation environment. OPNET simulations are time-based, so the user needs to specify the mean request interarrival time ($\frac{1}{\lambda}$, the mean service time per request ($\frac{1}{\mu}$) and the simulation duration $T$. OPNET's Discrete Event Simulator produces and consumes "requests" (more correctly, standard tokens) and records the queueing statistics requested by the user. OPNET request tokens are tagged by cluster ID and directed to a simulated

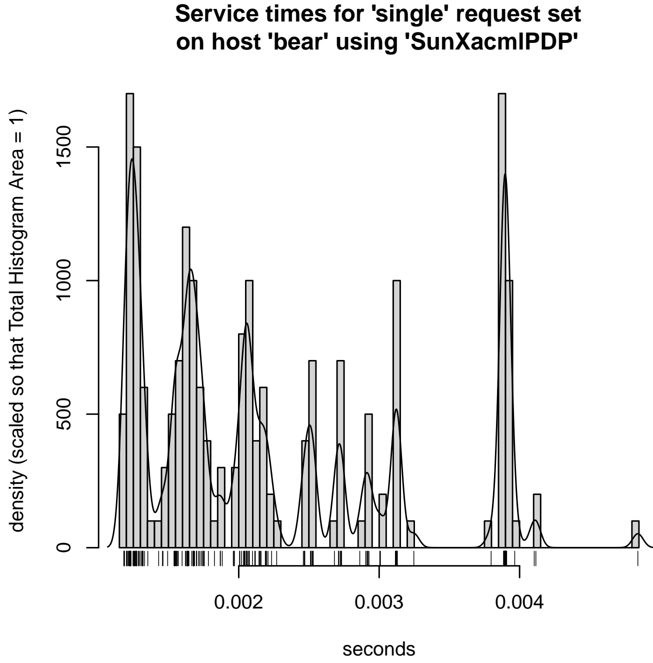



Fig. 4. Distribution of measured request service times on 'bear' using SunXACML.

server that handles one request at a time with the mean service time depending on the cluster ID tag, consistent with Figure 3.

## VII. Experimental Results

### A. Measured service times and clustering

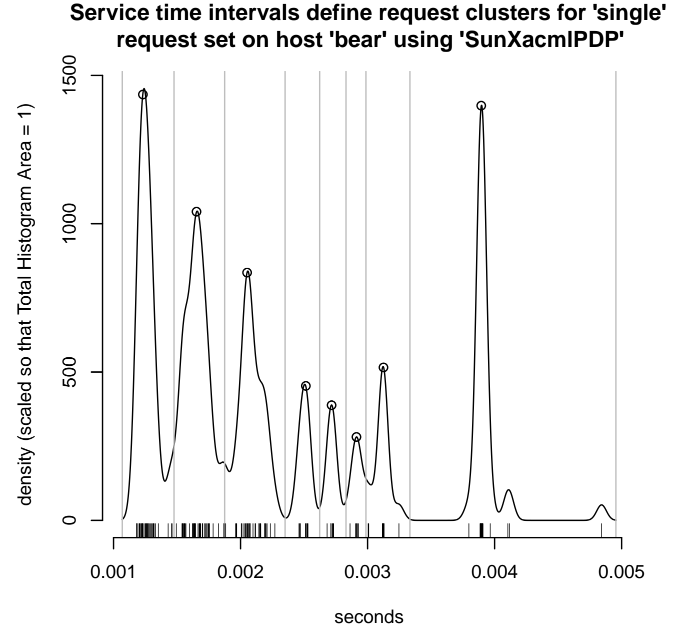
Figure 4 shows how service times are distributed for a given PDP-data combination. Plots like this alerted us to the presence of service time clustering. Referring to Figure 4, visual inspection suggests the number of clusters $n$ is 8 and the relative spacing tolerance is 0.05.

Figure 5a shows the clusters found by the algorithm described in Section IV when applied to the data in Figure 4. Clearly the clustering algorithm finds the main features in the service time data, though the cluster boundaries are not easily defined. For comparison, Figure 5b shows the equivalent clusters when Enterprise XACML is used instead of SunXACML PDP. In this case there are only 3 clusters, with most requests being assigned to the first cluster. The plots for cases using 'inisherk' instead of 'bear' and 'multi22' instead of 'single' are qualitatively similar to the Figures shown, indicating that the gross features (e.g., number of clusters and their sizes) of the service time distribution are determined by the PDP implementation.

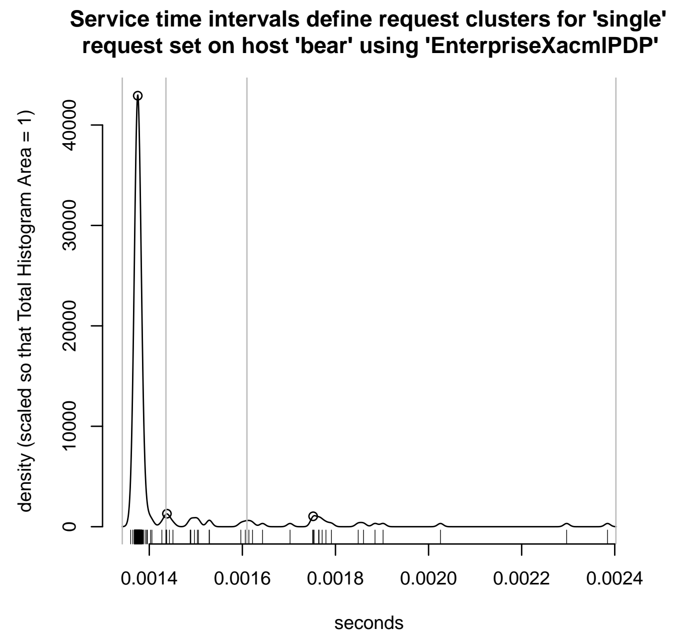
### B. Case study 1: Comparison

Given the experimental setup from Section VI and corresponding measurements from the testbed, namely

- the decision made by the PDP ('decision')
- request type (1 to 200, 'ind')
- the service time
- the cluster index




(a) Using SunXACML




(b) Using Enterprise XACML

Fig. 5. Clustering SunXACML and EnterpriseXACML request service times.

we can perform an Analysis of Variance to determine the contributions of factors and their interactions to the overall variance, see Table II. We note that all the identified factors, and their interactions, are very significant, except for the interaction between 'host' and 'reqGrp'.

Given this evidence that the ANOVA model appears significant, we proceed to an Analysis of Means. Reviewing Tables III and IV, 'inisherk' outperforms 'bear' and Enterprise XACML outperforms SunXACML, respectively.

| | Mean Sq | F value | Pr(>F) | Code |
|---|---|---|---|---|
| host | 2.8e-04 | 1910.4194 | 0 | *** |
| pdp | 4.2e-05 | 282.7925 | 0 | *** |
| reqGrp | 1.9e-06 | 13.2025 | 0.0002898 | *** |
| decision | 4.6e-05 | 313.5133 | 0 | *** |
| ind | 5.5e-07 | 3.7399 | 0 | *** |
| host:pdp | 5.9e-06 | 40.2613 | 2.999e-10 | *** |
| host:reqGrp | 5.3e-08 | 0.3609 | 0.5480907 | |
| pdp:reqGrp | 2.9e-05 | 195.0376 | 0 | *** |
| host:pdp:reqGrp | 2.7e-06 | 18.5428 | 1.778e-05 | *** |
| Residuals | 1.5e-07 | | | |

TABLE II
ANALYSIS OF VARIANCE RELATING (MEASURED) SERVICE TIMES TO EXPERIMENTAL FACTORS.

| | bear | inisherk |
|---|---|---|
| | 1.8e-03 | 9.5e-04 |
| rep | 800 | 800 |

TABLE III
COMPARISON OF SERVICE TIMES FOR HOSTS 'BEAR' AND 'INISHERK'.

Table V indicates that service times for 'multi22' requests are slightly less than those for 'single' requests, but more detailed study (i.e., white box testing) would be needed to discover why this might be true.

Interestingly, service mean times differ greatly by decision, with 'NotApplicable' decisions taking longer to make. This suggests that (some) PDPs might "fall through" to that decision only if other decisions are not available. It also suggests that there is a strong case for keeping policy sets up to date to avoid such (long service time) edge cases.

We present the 2-level interaction results in Tables VII, VIII and IX. Generally they confirm the overall main effects analysis above, but there is one anomalous result in that the mean service time for Enterprise XACML on 'inisherk' is greater than it is on 'bear'.

Summarising, collecting measurements from a balanced full factorial design such as this can provide insight into PDP performance because the researcher is able to control experimental conditions in the testbed.

| | SunXacmlPDP | EnterpriseXacmlPDP |
|---|---|---|
| | 1.5e-03 | 1.2e-03 |
| rep | 800 | 800 |

TABLE IV
COMPARISON OF SERVICE TIMES FOR PDPS 'SUNXACMLPDP' AND 'ENTERPRISEXACMLPDP'.

| | single | multi22 |
|---|---|---|
| | 1.4e-03 | 1.3e-03 |
| rep | 800 | 800 |

TABLE V
COMPARISON OF SERVICE TIMES FOR REQUEST GROUPS 'SINGLE' AND 'MULTI22'.

| | Deny | NotApplicable | Permit |
|---|---|---|---|
| | 1.3e-03 | 2.1e-03 | 1.1e-03 |
| rep | 1244 | 136 | 220 |

TABLE VI
COMPARISON OF SERVICE TIMES FOR DECISIONS 'DENY' AND 'NOTAPPLICABLE' AND 'PERMIT'.

| | | PDP | |
|---|---|---|---|
| | | SunXacmlPDP | EnterpriseXacmlPDP |
| host | bear | 2.01e-03 | 1.56e-03 |
| | inisherk | 1.05e-03 | 8.40e-04 |

TABLE VII
COMPARISON OF SERVICE TIMES FOR PDP:HOST INTERACTIONS.

### C. *Case study 2: Prediction*

In this scenario, we model the case where the PDP has reached a steady state ($\rho = 0.5$ is a constant), then 25% of request types suddenly have triple ($3\times$) their arrival rate, which is maintained over a prolonged period and then returns to its previous $\rho = 0.5$ level. Thus $\lambda^{(\text{overload})} = 0.25(3-1)\lambda^{(\text{base})} = 0.5\lambda^{(\text{base})}$, so the overload factor is $\gamma = 0.5$. While this is an idealised scenario, it might represent a situation where there is a sudden rise in access control requests on the hour as project groups attempt to initiate group chat sessions across a matrix-structured organisation.

To make the scenario more concrete, we need to choose how the additional requests are distributed across the clusters. We consider two such request distributions: *low* where the extra requests are skewed towards lower service times hence the lower clusters, and *high* where they are skewed in the opposite direction. For the free parameters in the model, we choose

$$\begin{aligned} \alpha_j^{(\text{overload:lo})} &= \frac{n-j+1}{\sum_{i=1}^{n} i} \\ \alpha_j^{(\text{overload:hi})} &= \frac{j}{\sum_{i=1}^{n} i} \end{aligned} \qquad (15)$$

Substituting Equations 14 and 15 in Equation 13 gives the required explicit expression for the overload process contributions $\rho^{(\text{overload:lo})}$ and $\rho^{(\text{overload:hi})}$.

| | | Request Group | |
|---|---|---|---|
| | | single | multi22 |
| host | bear | 1.83e-03 | 1.75e-03 |
| | inisherk | 9.70e-04 | 9.20e-04 |

TABLE VIII
COMPARISON OF SERVICE TIMES FOR REQUEST GROUP:HOST INTERACTIONS.

| | | Request Group | |
|---|---|---|---|
| | | single | multi22 |
| host | SunXacmlPDP | 1.70e-03 | 1.36e-03 |
| | EnterpriseXacmlPDP | 1.10e-03 | 1.30e-03 |

TABLE IX
COMPARISON OF SERVICE TIMES FOR REQUEST GROUP:PDP INTERACTIONS.

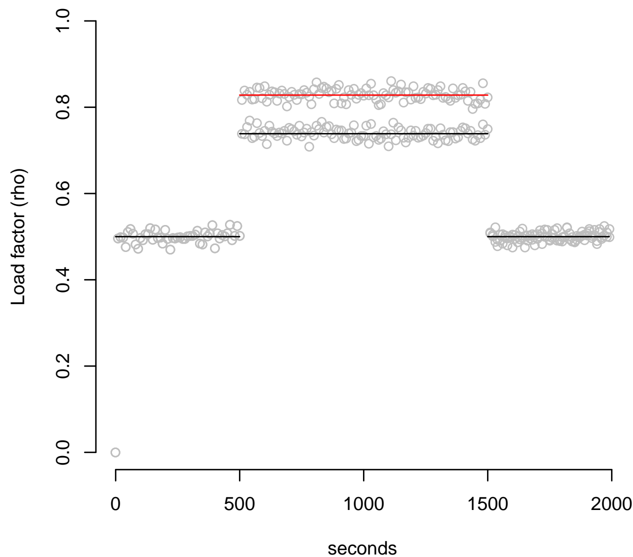



Fig. 6. Comparing server utilisation for 2 different overload request profiles. Unfavourable (mostly high service time) overload requests: $\max(\rho) > 0.8$. Favourable (mostly low service time) overload requests: $\max(\rho) < 0.8$.

The OPNET simulation model can also be extended to include overload arrival profiles equivalent to Equations 13. Note that the simulation (points) and explicit results (lines) in Figure 6 agree well and that the distribution of overload requests affects the overall load experienced by the PDP. Equivalent plots for Enterprise XACML showed smaller differences between the favourable and unfavourable overload request profiles, due to that PDP's different clustering behaviour.

## VIII. Summary and Future Work

We identified the performance bottleneck in XACML-based access control systems and built a measurement testbed to perform quantitative performance and scalability experiments. We collected timing measurements for a given policy set and associated requests and clustered the service times to create both a higher level analytical model and a more robust discrete event simulation. We considered two scenarios: 1) comparing two PDPs, studying the influence of the experimental conditions and 2) using the measurement clusters to *predict* performance for different overload conditions. Good agreement between analytic and simulated approaches was found, validating both approaches.

In future work, we wish to extend the testbed to introduce stochastic request arrivals (at present, requests arrive on a deterministic schedule). The resulting emulation data will facilitate comparison with the simulation results. We also wish to incorporate more realistic policy sets, request types and request arrival schedules and thereby to investigate more compelling scenarios.

## Acknowledgments

The authors acknowledge the contribution of Keith Griffin and Ger Lawlor, Cisco Systems Inc., who helped clarify the requirements for XACML server performance. The research was funded by the Science Foundation Ireland "FAME" SRC, grant 08/SRC/I1403 and by the EU FP7 Objective 1.4 Integrated Project ANIKETOS, ref FP7-257930.

## References

[1] T. Moses. (2005, February) eXtensible Access Control Markup Language TC v2.0 (XACML). OASIS. [Online]. Available: http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf

[2] D. Thakkar, A. E. Hassan, G. Hamann, and P. Flora, "A framework for measurement based performance modeling," in *WOSP '08: Proceedings of the 7th international workshop on Software and performance*. New York, NY, USA: ACM, 2008, pp. 55–66.

[3] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz, "Verification and change-impact analysis of access-control policies," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*. New York, NY, USA: ACM, 2005, pp. 196–205.

[4] S. Davy, B. Jennings, and J. Strassner, "The policy continuum-Policy authoring and conflict analysis," *Comput. Commun.*, vol. 31, no. 13, pp. 2981–2995, 2008.

[5] E. Martin, T. Xie, and T. Yu, "Defining and measuring policy coverage in testing access control policies," in *Proc. 8th International Conference on Information and Communications Security*, 2006, pp. 139–158.

[6] V. C. Hu, E. Martin, J. Hwang, and T. Xie, "Conformance Checking of Access Control Policies Specified in XACML," in *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 275–280.

[7] S. Marouf, M. Shehab, A. Squicciarini, and S. Sundareswaran, "Statistics & Clustering Based Framework for Efficient XACML Policy Evaluation," in *POLICY '09: Proceedings of the 2009 IEEE International Symposium on Policies for Distributed Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 118–125.

[8] P. L. Miseldine, "Automated XACML policy reconfiguration for evaluation optimisation," in *SESS '08: Proceedings of the fourth international workshop on Software Engineering for Secure Systems*. Leipzig, Germany: ACM, 2008, pp. 1–8.

[9] A. X. Liu, F. Chen, J. Hwang, and T. Xie, "Xengine: a fast and scalable XACML policy evaluation engine," in *Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 2008, pp. 265–276.

[10] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy, "Extending query rewriting techniques for fine-grained access control," in *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2004, pp. 551–562.

[11] H. Hamed and E. Al-Shaer, "Dynamic rule-ordering optimization for high-speed firewall filtering," in *ASIACCS '06: Proceedings of the 2006 ACM Symposium on Information, computer and communications security*. New York, NY, USA: ACM, 2006, pp. 332–342.

[12] V. Kolovski, J. Hendler, and B. Parsia, "Analyzing web access control policies," in *WWW '07: Proceedings of the 16th international conference on World Wide Web*. New York, NY, USA: ACM, 2007, pp. 677–686.

[13] F. Turkmen and B. Crispo, "Performance evaluation of XACML PDP implementations," in *SWS '08: Proceedings of the 2008 ACM workshop on Secure web services*. New York, NY, USA: ACM, 2008, pp. 37–44.

[14] B. Butler, B. Jennings, and D. Botivch, "XACML Policy Performance Evaluation Using a Flexible Load Testing Framework," in *Proc. 17th ACM Conference on Computer and Communications Security (CCS 2010)*. ACM, 2010, pp. 648–650, short paper.

[15] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2006, ISBN 3-900051-07-0. [Online]. Available: http://www.R-project.org

[16] L. Kleinrock, *Queueing Systems, Volume 1: Theory*. Wiley-Interscience, 1975.

[17] Z. Wang. (2010, February) Enterprise Java XACML. http://code.google.com/p/enterprise-java-xacml/wiki/DevelopmentPlan. Last accessed 2010-04-19.