# On the performance of access control policy evaluation

Leigh Griffin, Bernard Butler, Eamonn de Leastar, Brendan Jennings and Dmitri Botvich
Telecommunications Software & Systems Group,
Waterford Institute of Technology, Waterford, Ireland
{lgriffin, bbutler, edeleastar, bjennings, dbotvich}@tssg.org

***Abstract*—There is growing awareness of the need to protect digital resources and services in both corporate and home ICT scenarios. Meanwhile, communication tools tailored for corporations are blurring the line between communication mechanisms and (near) real-time resource sharing. The resulting requirement for near real-time policy-based access control is technically challenging. In a corporate domain, such access control mechanisms must be unobtrusive *and* comply with strict security objectives. Thus policy evaluation performance needs to be considered while addressing traditional security concerns. This paper discusses policy system design principles that motivate a novel Policy Decision Point (PDP) implementation and associated policy language. These principles are consistent with recent web development techniques designed to improve performance and scalability. Given a modern web development stack comprising a language (Javascript), a framework (Node.js) and a database management system (Redis), the proposition is that significant performance gains can be made. Our performance experiments suggest this is the case when, through various design iterations, our prototype PDP implementation is compared with an established, Java/XACML-based access control PDP implementation. The experiments presented in this paper suggest that newer technologies offer better performance. The analysis suggests that this is because they offer a more efficient data representation and make better use of computing resources.**

## I. Introduction

Information and communication technologies (ICT) have become indispensable in modern society. The technological and cost barriers to creating and sharing content are much lower than in previous decades. Direct communication between people is also easier, particularly when participants share their *presence* information. However, some communication events, with or without intermediate content, are considered "harmful". Consequently, access control *mechanisms* are used to prevent such harmful communication. For example

- A parent might wish to share family photographs with their relatives, but not with the wider user community of a photograph-sharing site;
- In an organisation, corporate governance procedures impose restrictions on staff using communication tools across groups, both internal and external.

Such scenarios require simple but robust and performant access control *procedures* that are informed by access control *policies*. Taking the latter scenario as an example, corporate integrated communication solutions [1], [2] have widespread adoption within industry. The tools have revolutionized communication by both enabling and controlling it, in near real time. As such, any access control mechanism used to protect corporation communications must provide a decision in near real time in order to preserve usability of the underlying communication medium. Much of the public literature on access control policies focusses on techniques to ensure that sets of deployed policies are consistent with high level security requirements. This paper focuses instead on how to optimize the policy evaluation performance at the Policy Decision Point (PDP) in order to meet performance objectives.

In previous work [3] we analysed the performance characteristics of two open source PDP implementations. Based on our analysis, we propose that the following features lead to improved PDP evaluation performance:

- Policies and requests should be encoded more efficiently
  - policies and requests should be relatively terse, to reduce the string handling overhead per request
  - policies and requests should be encoded in a way that minimizes the parsing overhead.
  - policies should be directly implementable.
- PDP implementations should be more efficient
  - policies and requests should be stored in ways that make retrieval more flexible and efficient
  - the PDP should scale outwards, to enable more efficient use of available resources.

This paper presents a careful analysis of the performance of different PDP implementations and considers the relationship between the PDP, the language that is used to encode the policies it uses and the access requests it decides upon. The first contribution is a novel PDP prototype implementation (labeled njsrPDP) that is motivated by the design principles above. It benefits from applying techniques that are achieving increasing attention in other domains [4], such as non-blocking I/O, to access control systems. Published open-source Java/XACML implementations do not employ such features. Unlike other proposals such as Xengine [5], the novel policy representation it uses is intended both to be easy for humans to interpret and to be as easy as XACML for machines to reason over. A parser was added to translate *existing* policies from XACML to its representation, therefore the new PDP (with its inbuilt parser) is (conceptually) a *drop-in replacement for a XACML PDP*. The second contribution is a new *representation* of policies (and requests) that takes full account of the performance improvements that are made

possible by the new PDP implementation. That is, it uses the same basic vocabulary and syntax as the translated policies and requests, but is optimized both to reduce unnecessary "bloat" and to make concept matching easier at policy evaluation time.

Section II examines related work in policy based access control and scalable web deployments. Section III describes our architectural decisions. Section IV discusses JSON and compares it with XML. Section V evaluates the prototype by comparing its performance against more established access control implementations. Section VI discusses the results of this evaluation. Section VII presents our conclusions.

## II. Related Work

The industry standard for access control policy specification and evaluation is XACML [6]. The standard architecture separates concerns effectively but practical deployments often encounter scalability difficulties, particularly in the PDP component. Researchers have begun to address the challenge of policy evaluation in XACML PDPs. We developed the STACS (Scalabity Testbed for Access Control Systems) testbed to enable performance experiments to be carried out under controlled conditions [7]. This testbed was used again to validate our hypothesis that the PDP implementation we propose has significant performance advantages, see Section V.

Butler et al. [3] surveyed proposals to improve the performance of policy-based access control. One of the most radical was proposed by Liu et al. [5]; the resulting Xengine PDP prototype evaluates policies and requests that were converted from textual to numerical format. The authors claim dramatic performance increases. However, the resulting representation is opaque, non-symbolic and difficult to analyse.

Most ICT resources requiring protection are made available using web standards. Thus it is instructive to see what those standards and practices might offer to access control deployments. In particular, cloud computing generates new and highly challenging requirements for scalability and elasticity. Non-blocking I/O, where interrupt handling is achieved through the extensive use of callbacks, has re-emerged as a possible solution. The need for blocking is removed by passing a callback parameter that is invoked on completion of the deferred task. Javascript [8], a language with the above characteristics, is being re-evaluated for use as a server-side language, with the community promoting the Node.js movement [9]. The result is better scalability, more efficient use of resources and very fast execution [10], [11]. More generally, software architects are exploring the use of the non-blocking I/O approach to re-appraise well-established software engineering best practices to tackle issues of scalability and performance. Previously, we investigated blocking and non-blocking approaches for high traffic real time services such as Instant Messaging [4], suggesting that non-blocking approaches have much better performance characteristics in that domain.

## III. Our architecture

With the Node.js framework [9], Javascript is no longer just a language supporting user interaction within browsers (client-side). Based on the Google initiated, open source V8 javascript engine, Javascript can be be compiled into highly optimized server-side machine code on the fly. The non blocking nature of Javascript is present within Node.js with all requests gradually executed in sequence through the usage of callbacks both in API design and usage. Node.js allows an elegant solution to be engineered for traditional scalability problems and an alternative approach for domains that might benefit from a non-blocking IO approach.

In addition to using Node.js, a complementary data persistence system is required. Maintaining the low-friction objective, a NoSQL database [12] with a Node.js interface was sought. NoSQL databases are non-relational, distributed databases that deliver horizontal scalability. Several options were considered, with each having bindings to Node.js ensuring compatibility. The data structure-oriented Redis server [13] was chosen because of its speed (in 1 minute, on one instance of the experimental platform, it averaged 11300 answered requests per second) and its data structures directly supported sets, lists and hashes, easing policy management.

Node.js provides a number of advantages that benefit the design of the access control system.

- It supports better modular design through component-based programming, benefiting from specifications supporting interoperability between server modules and even between server- and client-side modules;
- The non-blocking architecture facilitated by Javascript callbacks makes more efficient use of CPU resources than traditional systems that sit idle while a complex IO request is executing. In a Node.js deployment, the spare CPU cycles can be used more productively, e.g., to handle the next request or prepare the response;
- It uses events to trigger callback execution. A benefit of the event model is a publish/subscribe mechanism built into the environment. This promotes greater flexibility, since the access control system can include *listeners* (handlers) for particular events, such as *obligations*;
- The absense of complicated blocking semantics simplifies development.

The OASIS XACML TC defined data flows, see Figure 1, for XACML 2.0-conformant implementations. The flows required for a minimal installation of XACML were implemented in a Node.js supported Policy Execution Point (PEP), Policy Decision Point (PDP) and Policy Retrieval Point (PRP).

The other characteristics identified in Section I relate to implementation efficiency. The prototype PDP uses Redis to ensure flexibility and high performance when processing policy data structures, and the non-blocking I/O coding style fostered by the Node.js framework makes more efficient use of computing resources. The simplicity arising from streamlining policy evaluation should not be underestimated: the domain (policy) model is based on events, so if the execution model

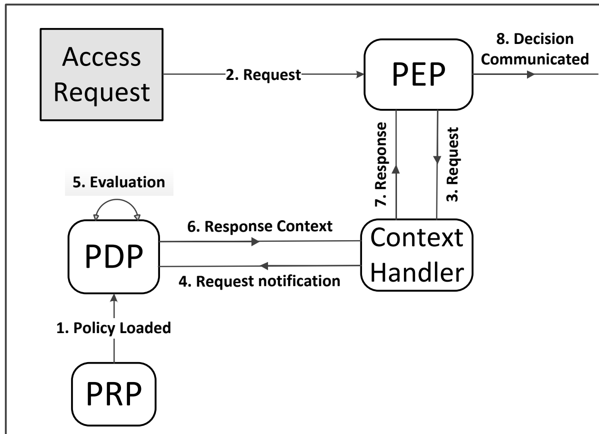



Fig. 1. XACML data flows and components relevant to this paper (a subset of Figure 1 in [6]).

is likewise, there is better alignment between the two. The PDP conforms to the CommonJS standard specification for module development [14]. This standard ensures the interoperability of a module and allows for integration with existing systems or future systems with syntax extensions. It also ensures that any Node.js modules will have the minimum features required to provide interoperability within the PDP. The PDP itself is composed of several functions governing both general and policy set specific behavior, mostly relating to style and formatting issues within XACML. The design is flexible with the ability to handle various policy and request combinations. This is evident throughout the njsrPDP scenario testing with multiple input types supported. The PDP is thus a fully independent generic PDP, with a core set of generic functions to interpret and successfully parse specific policy sets.

## IV. JAVASCRIPT OBJECT NOTATION

JavaScript Object Notation (JSON) [15] is a lightweight, text-based, language-independent data-interchange format derived from the object literals of the ECMAScript (JavaScript) programming language standard. JSON objects are analysed as string arrays, permitting higher parsing efficiency and easier preparation than heavier transport formats such as XML [16]. Crockford [15] describes JSON as “the fat-free alternative to XML” arguing that XML is not as well suited to data interchange because XML is much more verbose than JSON and rarely matches the data model of its host programming language. By contrast, JSON is built on just two data structures: a collection of name-value pairs and an ordered list of values. Having a data format that is interchangeable with a programming language’s builtin data structures eliminates translation time and reduces complexity and processing time. Furthermore, Wang [17] notes that the strengths of XML are also present within JSON, so nothing significant is lost.

### A. Conversion of existing XML policies or requests to JSON

Several attempts (such as [18]) to automate conversion between XML and JSON formats have emerged, thus enabling interoperability between XML policies/requests and a PDP that handles JSON natively. The conversion process makes structural changes to how data might traditionally be represented within JSON. For example, XML is designed as a language independent data representation format, which means metadata is associated with elements in order to ensure correct interpretation at a language level. Translation from XML to JSON indirectly brings this metadata, which is unnecessary within a JSON compliant language such as Javascript. The result is a “bloated” JSON format. Additionally, XML can contain sibling elements with the same outer identifier. However, JSON is a key:value storage mechanism, so it cannot have the same name for two keys. The solution, when translating, is the extensive use of arrays in JSON. While the JSON produced from a translation process is valid and semantically identical, it is harder to read and extra programmatic safeguards are needed to ensure correct interpretation.

```
"Policy":{
    "id":"RPSlist.7.0.1",
    "target":{
        "subjects":{
            "subject":{
                "role":"admin"
            }
        },
        "resources":{
            "resource":{
                "isPending":"false"
            }
        },
        "actions":{
            "action":{
                "action-type":"write"
            }
        }
    },
    "rule":{
        "id":"RPSlist.7.0.1.r.1",
        "effect":"permit"
    }
}
```

Fig. 2. JSONPL Policy Excerpt. The original XACML-encoded policy had 1473 characters versus 454 characters for the JSONPL encoding.

### B. JavaScript Object Notation Policy Language: JSONPL

Examining the output of the translated XML to JSON policy sets, the predefined vocabulary and semantics expressed in the XACML 2.0 specification [6] was apparent. Stripping away the redundant metadata and cleaning up the array translation process produced a policy vocabulary encoded in JSON that semantically was identical to the original XML policy. For brevity, the full formal JSONPL specification is not provided, however its major features are indicated in Figure 2. As with XML, deep nesting of arrays and objects is possible within JSON, allowing complex hierarchical structures to be represented. The dot notation is the most natural way to access data within a JSON document. For example, in Figure 2, the value associated with `role` can be accessed directly by calling `Policy.target.subjects.subject.role`, yielding the value `admin`.

The hierarchical structure of JSONPL mirrors that of XACML so a comparison is instructive. As with XQuery/XPath processing of XML documents, the execution time for retrieving a value from a known location in a JSON data structure is independent of the size of that structure. Policy and/or rule combining algorithms can be applied in JSONPL in the same way as in XACML. The policy language is as expressive as an XML based XACML policy, is arguably more human-readable and provides native compatibility with many programming languages, easing authoring and interpretation issues. To the best of our knowledge, a JSON-based access control policy language has not been attempted before and as such represents a novel contribution. All the desirable features relating to policy and request specification and encoding identified in Section I are provided. As seen in Section V, the policy language supports rapid response times with low overheads in a suitable PDP.

# V. Evaluation

## A. Comparison of PDPs

Experiments were performed to compare the JSON/Node.js/Redis implementation described above with more traditional XACML/Java implementations of SunXACML and EnterpriseXACML. A set of XACML policies and their related requests was chosen and were translated manually to their JSONPL equivalents. The two Java-based PDP implementations were placed in STACS [3] so that service times per request could be recorded in a repeatable fashion. The prototype Node.js implementation was instrumented in the same way, taking advantage of the Node.js eventing model to collect service times based on the same triggering events that were used in STACS:

- PDP Policy Read *start*
- PDP Policy Read *end*
- Request *arrives at* PDP
- Response *leaves* PDP.

A simplified queueing discipline was employed, namely, when response $n$ from the PDP arrived at the PEP, it triggered the submission of request $n+1$ from the PEP to the PDP. This sequential processing was easily achieved in STACS using loops and in the njsrPDP harness using callbacks. The entire experiment was replicated $N_{\text{rep}} = 100$ times, in random order, for each set of `host` $\times$ `pdp` conditions. When measuring elapsed times in the PDP, we do not have full control over other processes that can use computing resources needed by the PDP. Thus the measured service times generally have a positive bias. More formally, assume that service time $\hat{t}$ measured by $t_i, i = 1, \ldots, N_{\text{rep}}$ is subject to additive nonnegative error $e_i$ according to

$$t_i = \hat{t} + e_i, \text{ where } e_i \geq 0, \ i = 1, \ldots, N_{\text{rep}}. \quad (1)$$

The best estimate of $\hat{t}$ is given by $\min_i\{t_i\}, \ i = 1, \ldots, N_{\text{rep}}$.

The measured service time data was standardized to use the same labels and time units to ensure that data features were consistent between STACS and non-STACS sources.

TABLE I
Service time measurements and their context.

| Name | Type | Possible values |
|---|---|---|
| policy | Common | `continue` |
| reqGrp | Common | `single` |
| host | Factor | `bear`, `inisherk` |
| pdp | Factor | SunXACML, EnterpriseXACML, njsrPDP |
| duration | Response | Numeric |

TABLE II
Scenario conditions

| | Manually generated Policies | Auto generated policies (bloated) |
|---|---|---|
| Manually generated Requests | Scenario 1a | Scenario 1b |
| Auto generated requests (bloated, prepared) | Scenario 2a | Scenario 2b |
| Auto generated requests (bloated, on the fly) | Scenario 3a | Scenario 3b |

The factors considered in our main experiment are shown in Table I. The `continue` policy and `single` request group are published (in XACML form) as part of the test suite for XEngine [5], and were translated to JSON format as described earlier. This policy set and associated requests was used in the experiments and models access control rules and requests for a Conference Paper Management System. While that domain does not require microsecond evaluation times, the policy set contains reasonably complex business rules such as separation of duties constraints and other features representative of real-time corporate communications. The two `host` instances are Intel 64-bit dual-core machines, each with 2GB RAM but differing in other computing resources, running Ubuntu 11.04.

The primary experiment compares njsrPDP with two existing XACML PDP implementations. The secondary experiment examines *how* njsrPDP achieves increased performance.

## B. Experimental Scenarios

Six experimental scenarios are considered as described in Table II and were used to compare the effects of different policy and request formulations for a given PDP (in this case, the njsrPDP prototype).

Referring to Figure 3, we see the main features of the scenarios to be compared with each other. Summarising,

- the A scenarios formulate the *policies* as JSON in ways that are "optimized" for evaluation performance, while the B scenarios use the policies as translated by a generic XML to JSON converter.
- the "1" scenarios formulate the *requests* as JSON in ways that are "optimized" for evaluation performance
- the "2" and "3" scenarios use the requests as translated by a generic XML to JSON converter. They differ in respect of when the translation occurs: *before* reaching the PEP for "2", or within the PEP itself for "3".

Note that the experimental conditions in Scenario 1A are those used when comparing njsrPDP with the SunXACML and EnterpriseXACML PDPs, see Section V-A.

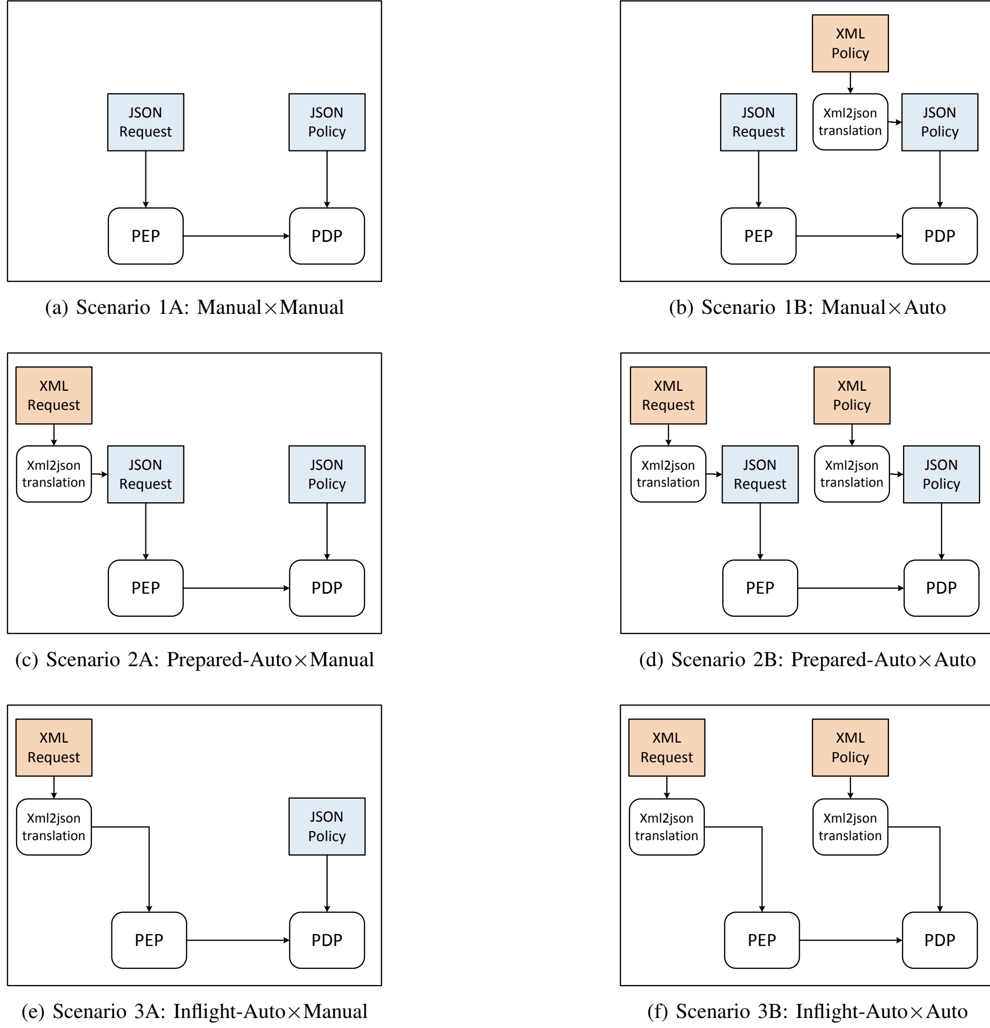



Fig. 3. njsrPDP policy×request scenarios; scenario conditions are defined in Table II.

In summary, all of the policies and request artefacts were originally encoded as XACML and so benefits from the XACML ecosystem. However the artefacts are converted to JSON by different methods and at different stages of policy evaluation. The performance improvements arising from each research contribution can be estimated by comparing the timing results.

## VI. RESULTS

### A. Comparing *njsrPDP* with its peers

Figure 4 shows histograms of the service times for SunXACML, a *reference* Java-based XACML PDP, compared with the service times for njsrPDP, the implementation introduced in this paper. The influence of the `host` and `pdp` factors can be seen clearly. Indeed, the new implementation has noticeably better performance when other factors are equal.

The Node.js/Redis prototype PDP implementation, labeled njsrPDP in Figure 5 has the following performance features:

1) The mean service time per request is much less (one

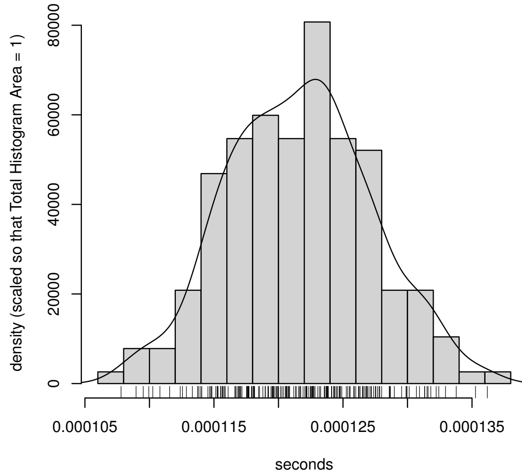
TABLE III
ANALYSIS OF VARIANCE: HOST, PDP, HOST:PDP EFFECTS ARE VERY SIGNIFICANT—$\alpha$ PROBABILITY UNDERFLOWS MACHINE EPSILON $\varepsilon$.

| | Df[a] | SumSq | MeanSq[b] | F value[c] | Pr(>F) |
|---|---|---|---|---|---|
| host | 1 | 1.97e-05 | 1.97e-05 | 2.24e+04 | $< \varepsilon$ |
| pdp | 2 | 1.15e-04 | 5.75e-05 | 6.55e+05 | $< \varepsilon$ |
| decision | 2 | 1.00e-09 | 5.00e-10 | 5.14e-01 | 0.60 |
| requestIndex | 190 | 1.61e-07 | 1.00e-09 | 9.67e-01 | 0.61 |
| host:pdp | 2 | 7.50e-06 | 3.75e-06 | 4.27e+03 | $< \varepsilon$ |
| Residuals[d] | 954 | 8.37e-07 | 1.00e-09 | | |

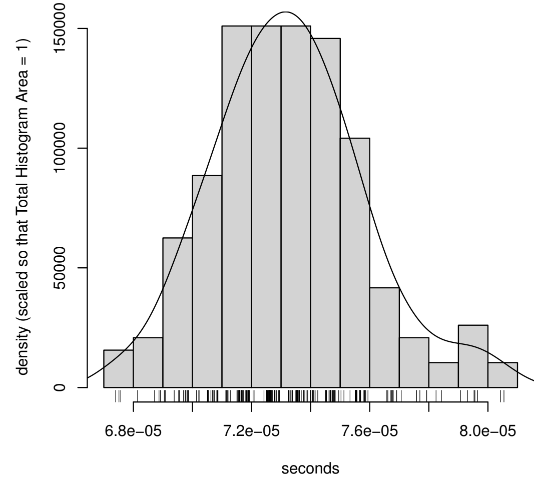
[a](Number of) degrees of freedom
[b]SumSq/Df
[c]F ratio: MeanSq/MeanSq_Residuals
[d]Other, unspecified factors

TABLE IV
ANALYSIS OF MEANS: HOST `INISHERK` HAS BETTER PERFORMANCE THAN `BEAR`.

| host | bear | inisherk |
|---|---|---|
| time | 6.3e-04 | 3.7e-04 |
| #replicates | 576 | 576 |




(a) njsrPDP on bear




(b) njsrPDP on inisherk

Fig. 5. njsrPDP request service times on hosts `bear` and `inisherk`.

TABLE V
ANALYSIS OF MEANS: PDP NJSRPDP HAS BETTER PERFORMANCE THAN THE OTHER PDPS.

| pdp | SunXACML | EnterpriseXACML | njsrPDP |
|---|---|---|---|
| pdp | 5.56e-04 | 8.61e-04 | 9.3e-05 |
| #replicates | 384 | 384 | 384 |

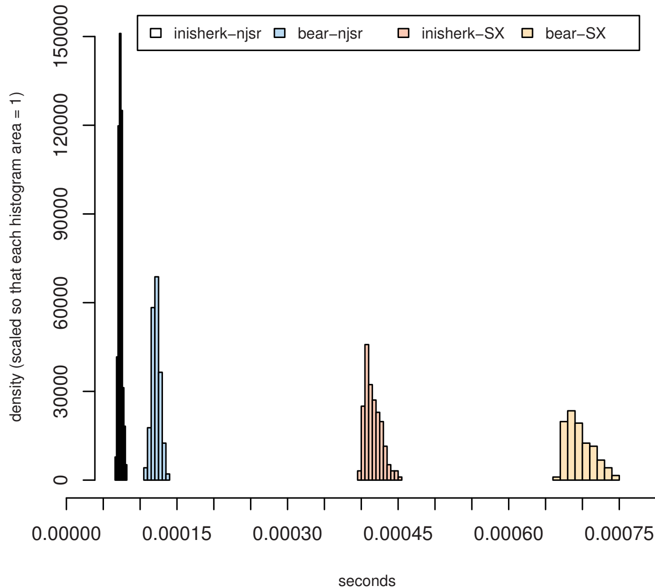



Fig. 4. Comparative service time histograms for hosts `bear` and `inisherk` and PDP implementations SunXACML and njsrPDP, for Scenario 1A.

sixth that of SunXACML, one eighth that of EnterpriseXACML), see Table V.

2) The performance profile for njsrPDP is bell-shaped; for EnterpriseXACML it is approximately uniformly distributed; for SunXACML it is a skewed mixed distribution.
3) The implementation on the two hosts shows a similar profile (see Figure 5) though different performance levels, see Table IV because `inisherk` has a faster CPU and more L1 cache. This suggests that performance scales vertically on a single host and also that the performance profile and observations are *reproducible*.

It should be noted from Table III that these differences are statistically significant [19] and hence are highly unlikely to arise by chance. The challenge is to show how the design principles outlined in Section I and implemented in the JSONPL prototype described in Section IV contribute to the statistically significant performance improvements summarized in Table V.

The system resources used by the JSON and XACML implementations were captured using `dstat`, which collects resource statistics (cpu, memory, disk usage, etc) on a timed basis while the experiments run in the testbed. Figure 6 shows that njsrPDP uses far less CPU (10% versus 60%, say). The cpu `wait` time is generally low, suggesting that both Node.js and the JVM are quite efficient. However, the `user` cpu cycles are much greater for the Java/XACML implementations. The CPU has to work much harder to evaluate policies in Java/XACML PDP implementations. This is consistent with observations elsewhere in building scalable web applications [4]. Furthermore, the `idle` cpu usage is much higher for njsrPDP, suggesting there is much more capacity available for increased throughput.

The memory usage was also recorded, supporting the contention that njsrPDP makes particularly efficient use of computing resources, including 35% less memory.

### B. Policy-Request Scenario Comparison

Scenario 3 incurs serious overheads. Firstly, the converter requires 83% of the total time needed to make the access decision. Secondly, translation introduces a large object that needs to be maintained at the top of the callback chain. When the PDP evaluates and wishes to pass its decision to the PEP it

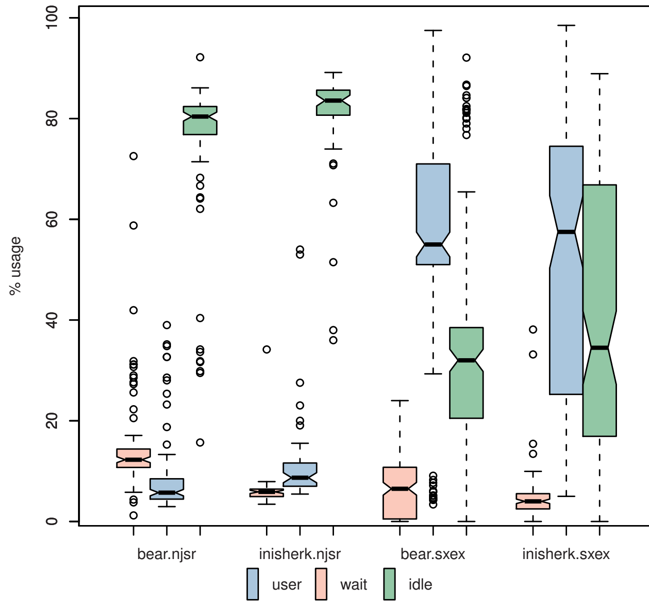



Fig. 6. CPU usage for selected host × pdp combinations. Hosts are `bear` and `inisherk` and PDPs are SunXACMLEnterpriseXACML (`sxex`) and njsrPDP (`njsr`).

must "walk" back up the callback chain. The top level callback needs to retain a link including the context of the request and its arguments throughout the whole chain. By placing such a large object in the top callback, translation imposes greater overheads down the callback chain, so the computation time is increased. Therefore the next step is to investigate how the system would perform if the penalty for translation, which increases evaluation time in two ways, were removed.

By pretranslating the requests the overhead incurred in translating the requests at run time is removed as well as the added overheads in the callback chain. The challenge becomes that of guaranteeing safe and accurate policy evaluation. Some approaches, identified in Section IV, impose performance losses while traversing arrays, with other losses emerging when developing the PDP. One complication is that, depending on the XML schema, the ordering of some child elements may be unspecified. Consequently the position of sibling child elements in policies within the same policy set can be different. A XACML PDP's XML parser has no difficulty in this regard but the translating program makes no allowance for consistency in the generated JSON. Thus njsrPDP has to account for this, handling all ordering permutations so as to operate correctly. While these problems also occur in Scenario 3, the additional translation overhead masked this feature. A minor performance gain was identified when using a combination of pre translated JSON requests and optimized (JSONPL-formatted) policies, as there is less overall bloat.

The boxplot in Figure 7 indicates that the best performance is obtained when optimized (JSONPL) policies and requests are used (Scenario 1A) and that performance degrades as bloated/more complex automatically translated JSON is used to represent polices and requests (Scenario 2B).

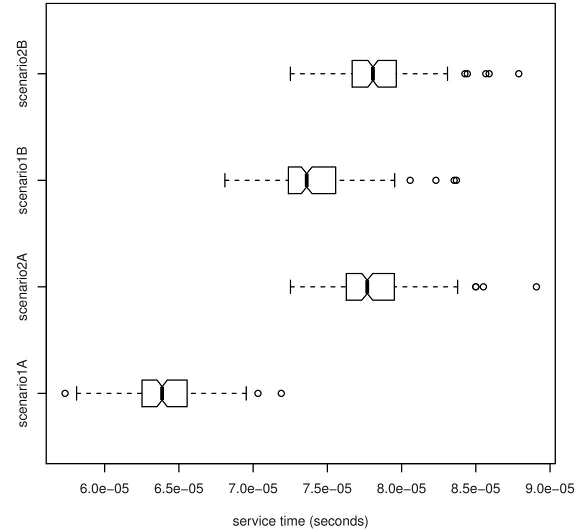



Fig. 7. Service times for Scenarios 1A, 1B, 2A, 2B

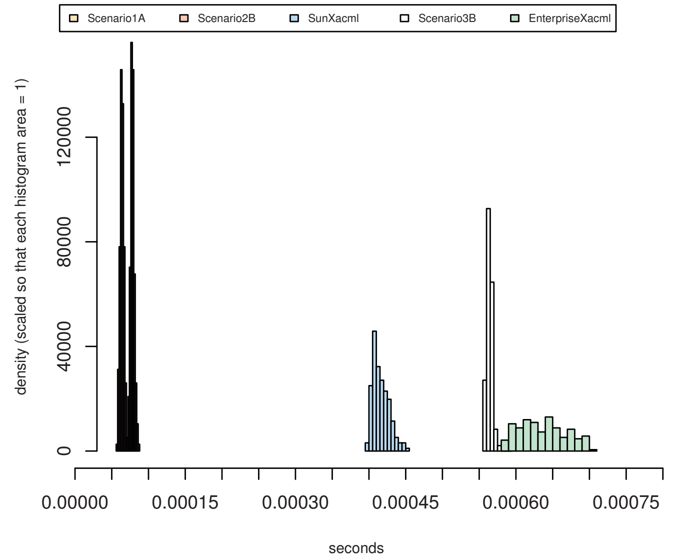



Fig. 8. Service time comparison. Ranked in decreasing order of performance (left to right in the figure above), they are: njsrPDP Scenario 1A, 2B; SunXACML; njsrPDP Scenario 3B, EnterpriseXACML.

The service time histograms in Figure 8 show how different njsrPDP scenarios compare with "traditional" PDP implementations. Clearly there is no net performance benefit of the JSON implementation when requests are translated on the fly: mean services times for njsrPDP Scenario 3A are greater than those for SunXACML, but njsrPDP has much greater performance potential (see Scenario 1A).

Table VI confirms that factors such as scenario type, decision and request type are significant variables when modelling service times. The comparison of mean service times for each Scenario in Table VII shows how different Scenario 3 is to

TABLE VI
ANALYSIS OF VARIANCE FOR SCENARIO SERVICE TIMES

| | Df | Sum Sq | Mean Sq | F value | Pr(>F) |
|---|---|---|---|---|---|
| scenario | 5 | 6.4e-05 | 1.27e-05 | 7.43e+05 | $< \varepsilon$ |
| decision | 2 | 1.0e-09 | 4.00e-10 | 2.54e+01 | 1.8e-11 |
| requestIndex | 190 | 8.0e-09 | 0.00e+00 | 2.32e+00 | $< \varepsilon$ |
| Residuals | 954 | 1.6e-08 | 0.00e+00 | | |

TABLE VII
MEAN SERVICE TIMES FOR EACH OF THE SCENARIOS

| | | | |
|---|---|---|---|
| *1A* | 6.4e-05 | *1B* | 7.4e-05 |
| *2A* | 7.8e-05 | *2B* | 7.8e-05 |
| *3A* | 57.8e-05 | *3B* | 56.5e-05 |

the others, and how much request format optimization affects performance (compare Scenario 1A and 2A).

## VII. CONCLUSIONS

We outlined why access control systems are necessary, and why policy evaluation performance is a particular concern. We suggested some features that are likely to improve PDP performance and that motivated us to consider whether implementations based on new languages, transfer formats, frameworks and storage mechanisms might benefit policy evaluation performance. Thus we reviewed recent progress in frameworks, languages and practices for developing scalable web services more generally and showed how they are consistent with the design features we identified as being desirable for policy evaluation performance.

The policy evaluation service times obtained using njsrPDP are significantly less than those of traditional approaches, supporting the analysis in Section V. This performance improvement was achieved without compromising the semantics of the policy set or its readability—arguably, the JSONPL encoding is easier for humans to understand.

Based on its performance results, the prototype njsrPDP shows considerable promise. The authors published some of the modules as open source components [20] to help it achieve its potential. The published components also include policies and requests in XML, JSON and JSONPL formats.

In the future, we wish to test the performance of njsrPDP with more extensive policy sets. This will enable us to measure the effects of performance enhancements and configuration changes and serve to validate the performance benefits in typical scenarios. Ensuring compatibility with existing policy based specifications, notably XACML, warrants further investigation. By automating the process of translating an XML file into formatted JSONPL, and vice versa, the potential of the underlying technological stack would be maximized. This would be a challenging but reachable goal. Inside, extended, semantically aware converters would allow backwards compatibility with existing legacy policy systems and potentially allow JSONPL to act as a DSL for policy authoring. The *performance* results in this paper are impressive, but further experiments are needed to test whether they result in greater *scalability* and *elasticity*. Furthermore, we assume that high-level "business" policies of the policy continuum [21] have already been converted to lower level policies referencing managed entities directly. Thus the wide gap between high- and low-level policy specifications, which is an important research topic in its own right that may also affect policy evaluation performance, is not in scope for this paper.

## ACKNOWLEDGMENT

The authors acknowledge funding from Science Foundation Ireland grant 08/SRC/I1403 "Federated Autonomic Management of End-to-End Communication Services" and from the Irish HEA PRTLI Cycle 4 FutureComm programme.

## REFERENCES

[1] Cisco. (2012, Jan) Voice and Unified Communications. [Online]. Available: http://www.cisco.com/en/US/products/sw/voicesw/index.html
[2] IBM. (2012, Jan) Unified Communications. [Online]. Available: http://www.ibm.com/software/products/us/en/category/SWAAA
[3] B. Butler, B. Jennings, and D. Botvich, "An experimental testbed to predict the performance of XACML Policy Decision Points," in *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, Dublin, Ireland, 2011, pp. 353–360.
[4] L. Griffin, K. Ryan, E. de Leastar, and D. Botvich, "Scaling Instant Messaging communication services: A comparison of blocking and non-blocking techniques," in *IEEE Symposium on Computers and Communications (ISCC)*, 2011, pp. 550–557.
[5] A. X. Liu, F. Chen, J. Hwang, and T. Xie, "Xengine: a fast and scalable XACML policy evaluation engine," in *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. New York, NY, USA: ACM, 2008, pp. 265–276.
[6] OASIS XACML-TC. (2005, February) XACML 2.0. [Online]. Available: http://docs.oasis-open.org/xacml/2.0/XACML-2.0-OS-ALL.zip
[7] B. Butler, B. Jennings, and D. Botvich, "XACML Policy Performance Evaluation Using a Flexible Load Testing Framework," in *17th ACM Conference on Computer and Communications Security (CCS 2010)*. ACM, 2010, pp. 648–650, short paper.
[8] D. Crockford, "ECMAScript Language Specification," Jun. 2011. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-262.htm
[9] R. Dahl, "Node.js: Evented IO for V8 javascript," Aug. 2011. [Online]. Available: https://github.com/joyent/node
[10] R. M. Lerner, "At the forge: Node.JS," *Linux J.*, vol. 205, May 2011.
[11] S. Tilkov and S. Vinoski, "Node.js: Using JavaScript to Build High-Performance Network Programs," *IEEE Internet Computing*, vol. 14, no. 6, pp. 80 –83, Nov–Dec 2010.
[12] R. Cattell, "Scalable SQL and NoSQL data stores," *SIGMOD Rec.*, vol. 39, pp. 12–27, May 2011.
[13] R. M. Lerner, "At the forge: Redis," *Linux J.*, vol. 197, Sep 2010.
[14] CommonJS, "Commonjs modules 1.1 specification," Jan. 2012. [Online]. Available: http://wiki.commonjs.org/wiki/Modules/1.1
[15] D. Crockford. (2006) JSON: The Fat Free Alternative to XML. 15th International World wide Web Conference. WWW 2006. Edinburgh, Scotland. [Online]. Available: http://www2006.org/programme/item.php?id=d8
[16] S. Downes, L. Belliveau, S. Samet, and R. Abdur Rahman, M. and-Savoie, "Managing Digital Rights Using JSON," in *7th IEEE Consumer Communications and Networking Conference (CCNC)*, 2010, pp. 1–10.
[17] G. Wang, "Improving Data Transmission in Web Applications via the Translation between XML and JSON," in *3rd IEEE International Conference on Communications and Mobile Computing*, ser. CMC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 182–185.
[18] BugLabs, "Node xml2json," Jan. 2012. [Online]. Available: https://github.com/buglabs/node-xml2json
[19] P. Dalgaard, *Introductory Statistics with R*, ser. Statistics and Computing. Springer, 2008.
[20] L. Griffin, "JSONPL repository," Aug. 2011. [Online]. Available: https://github.com/lgriffin/JSONPL
[21] S. Davy, B. Jennings, and J. Strassner, "The policy continuum—policy authoring and conflict analysis," *Computer Communications*, vol. 31, no. 13, pp. 2981–2995, August 2008.