An Access Control Implementation Targeting Resource-constrained Environments

Fan Zhang, Bernard Butler, Brendan Jennings
Telecommunications Software & Systems Group, Waterford Institute of Technology, Ireland

***Abstract*— As more and more services are deployed on devices near the network edge, security operations (such as authentication and authorization) need to move with them. Typically, edge devices have fewer resources than data center servers and so the security operations need to make more efficient use of what is available while offering adequate performance. Authorization adds latency and requires system resources, but the need for security management with strong authorization at the network edge is growing. We have released the *first* open source, high-performance, resource-efficient, XACML3 standard-compatible Policy Decision Point (PDP) called `Luas` (means "speed" in the Irish language) based on an event-driven architecture and a non-blocking computational model, using a *Bloom Filter* for better performance. We compared its performance, resource usage and reliability against existing open source PDPs. Like those we tested, it provides accurate decisions, but `Luas` offers much faster security policy evaluation while using fewer system resources, and provides responses in a reasonable timeframe even when resources are scarce.**

I. Introduction

Managing network and computational resources for security operations is a difficult task because their needs are not always well understood, compared to other operations that draw upon the same finite resources. With growing regulation and awareness of the risks of poor security hygiene, security operations are now more important than ever. They address requirements such as security and privacy. Authentication is usually a one-off concern per session, but authorization is ongoing and hence arguably requires more management attention.

Data security concerns are growing. They present a series of access control challenges that include data privacy breaches, unintended and/or malicious updates/deletion of data and other threats to data availability, such as those posed by ransomware. Increasingly, the network edge is seen as the new security frontier and needs to be managed as such. This management task is challenging because of high complexity (e.g., composing and processing many data flows) and low resource availability (most edge devices are small, battery-powered and dedicated to other tasks). We address the issue of limited resources for security operations in this paper.

To ensure access controls are applied consistently, all requests to use resources must be checked against the relevant access rules. However, this check adds latency to the primary operations of the system, and so might cause unacceptable performance bottlenecks. Thus, *high-performance*, *resource-efficient* security operations are essential to ensure adequate overall system performance and to use as few resources (CPU, memory, network bandwidth) as possible, so that they are available for other operations. XACML is one of the most widely used languages for expressing complex access control policies [1]. A performance bottleneck might occur when access requests are sent to a Policy Decision Point (PDP) at very high rates, particularly where state changes occur and the decision depends on dynamically changing context. Meanwhile, the security operations collectively provided by the PDP, PEP, etc., can require significant resources. Therefore, there is a need for high performance, scalable and reliable PDP and supporting server infrastructure, regardless of where the hardware and software components are placed in the network.

This paper makes two key contributions. First, a standard-compliant high-performance event-driven XACML PDP implementation is developed and released as open-source. This implementation is packaged as a JavaScript module and available in Node Package Manager (NPM). NPM is the package manager for JavaScript and the world's largest software registry. Integrating into the Node.js ecosystem is trivial using 'npm install luas' so it can be a dependency of another Node.js application or web applications. Our PDP is not only able to provide an attractive option for those building systems that need to meet strong security and privacy guarantees but also maintains high reliability and accuracy. Second, we proposed an approach that applies *Bloom filters* to policy evaluation, enabling the PDP to match, with low memory usage and minimal delay, the request against a policy and rule.

Section II describes previous work on PDP performance improvement. Section III motivates the technological choices (primarily the paradigm, platform and language) made when developing **`Luas`**. Section IV describes experiments where **`Luas`** was compared against its peers: how these were chosen, the test framework and scenarios. Section V presents our results, which indicate that **`Luas`** meets its objectives. Finally, Section VI presents our conclusions and possible future work.

II. Related Work

In order to generate an access decision for a request, the policy decision point (PDP) needs to parse the corresponding policy sets and to evaluate the relevant rules that are defined in the policy set for that request. In previous work [2] we warned that the PDP could become a performance bottleneck and built a comprehensive testbed that can be used to carry out performance experiments while controlling resource usage.

Various approaches were proposed to improve authorization performance. Jahid et al. [3] convert high-level attribute-based policies into Access Control Lists for resources in a

978-3-903176-24-9 © 2019 IFIP

database. Their model performed better than the established SunXACML reference PDP implementation.

Liu et al. [4] proposed XEngine, a new XACML PDP written in Java, to evaluate policies more efficiently. Their improvements include attribute numericalization and normalized structure. XEngine had much lower request processing time than the reference SunXACML PDP.

Mourad et al. [5] developed the SBA-XACML PDP using the PHP language. It uses set-based algebra to match the policies. SBA-XACML was 3.2 times faster than XEngine and 8 times faster than SunXACML, for single-valued requests.

Pina Ros et al. [6] proposed 'Graph-Based' models for evaluating policies based on 'Matching Tree' and 'Combining Tree' structures. Their approach was faster than the SunXACML reference PDP. Ngo et al. [7] noted that XEngine cannot be used with all policies, e.g., where the attribute conditions are not equalities. Their sne-xacml PDP implementation is more general than either XEngine or the Pina Ros et al. [6] PDP, but also has very good performance.

Each of these papers optimised the performance of XACML policy evaluation. Their approaches are similar because they improved the performance by adopting a tree, a graph or similar data structure. However, they support simpler policy models with reduced semantics, so all fall short of full compatibility with the XACML 3 standard. The only publicly announced full-XACML PDPs (SunXACML, Enterprise XACML, Balana and AT&T XACML) all employ *full enumeration* and so are measurably less performant than those PDPs that sacrifice full XACML semantics for performance.

Earlier performance evaluation efforts such as Butler and Jennings [8] and Turkmen and Crispo [9] focused on per request performance: how long does it take each PDP implementation to evaluate a given access request. Furthermore, [8] provided *recommendations* relating to design and deployment strategy, to improve evaluation performance, so that the resulting security operation is less likely to be *CPU-bound*. However, access control evaluation forms part of a larger system. The arrival pattern is typically bursty, so that even PDPs with good computational performance can become *I/O-bound*; this was not considered directly before. Also, previous papers comparing PDP performance largely ignored the system resource requirements for each PDP under test.

III. Design and Implementation of **Luas**

The current best practice for web server development favours the use of an *Event-Driven* programming paradigm with a *non-blocking* I/O model, where the execution flow is determined by the events. We decided to investigate this approach.

McCune [10] compared three simple file processing programs in JavaScript, Ruby, and Java. Each was hosted on its corresponding web server (Node.js, EventMachine, Apache). A client sent requests to those web servers and the programs processed files in response. The results showed that the Node.js server performed better (more requests processed per second; number of files opened simultaneously) than either of the other two servers.

A. Platform & Language Selection

We reviewed the existing PDP implementations and identified the following criteria as being desirable when choosing a programming language to develop an efficient PDP implementation

- Able to process high-volume workloads, e.g., when the system becomes I/O-bound
- High system resource efficiency
- Friction-less, enabling one runtime to serve the PDP and other applications
- Support for modern development practices
- Large community support

Based on these criteria, we compared C++, Java, Rust and JavaScript in terms of their development platforms, runtime environments and libraries. C++ has a steep learning curve and lacks built-in memory management. Although Java has been widely used to implement PDPs and Java NIO enables non-blocking I/O, Java is not asynchronous in spirit and implementing non-blocking applications with NIO is quite complex. The Rust version we reviewed did not have non-blocking I/O, nor a strong ecosystem to support development. We concluded that the best candidate currently is JavaScript with the Node.js platform for server-side operation.

B. Policy Evaluation Approach

The PEP constructs a request from its *Subject*, *Resource*, *Action* and *Environment* attributes and sends it to the PDP. The PDP needs to find the rule(s) that best match the attributes in the request, so it extracts the request attributes and finds one or more applicable policies based on matching those attributes to the corresponding elements in the policy *Target*s (policy or rule matching). When an applicable Target is found, the PDP applies the associated rule, which can also include a filtering *Condition* expressed as a complex Boolean expression. If the condition is fulfilled, the *Effect* (either *deny* or *permit*) will be returned. In cases where more than one policy or rule is matched, the final access decision depends on the relevant *Combining Algorithm* to resolve conflicts among matched policies or rules.

We found that existing standard-compliant PDP implementations spend most of their time matching attributes, given large policies. We believe their use of brute-force search is one of the reasons: existing PDPs iterate through all the attribute categories in a *Target*, so they undertake more work than is strictly needed. We propose a rapid and memory-efficient policy and rule searching technique that uses Bloom Filters.

The Bloom filter [11] is a probabilistic query data structure, which is designed to test the existence of an element in a data set rapidly and space-efficiently. Its data structure is a *Bit Vector* B with m elements, with an initial value of zero for each element. The Bloom Filter requires k independent hashing functions, $\{\hbar_j\}; j = 1, \ldots, k$, each hash function having the range $\{1, \ldots, m\}$.

As an example, suppose we have a set $X = \{x_i\}$ with n elements, and a possible element of that set y, and we wish to test whether $y \in X$. To compute the bit vector $B(X)$, we generate $\{\hbar_j(x_i)\}$ for $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, k\}$, then the bits in B at positions $\hbar_j(x_i)$ in B are set to 1 for all $x_i \in X$ and $j \in \{1, \ldots, k\}$. Note that a bit b_i can be set to 1 multiple times, if more than one of those hash functions evaluates to b_i. Similarly, we can compute $B(y)$. If $B(X)$ has 1 in every position where $B(y)$ has 1, which is equivalent to asking whether $B(X) \wedge B(y) = B(y)$, then return True (there is *probably* a match), otherwise return False (there is *certainly* no match).

Because hash functions are used, collisions can occur, so the Bloom Filter is approximate: we can be certain if no match occurs, but there is a small probability (approximately $\left(1 - e^{-kn/m}\right)^k$ of a false positive (i.e., of deciding $y \in X$ when it is not). The false positive rate increases with n (the size of X) and decreases as both k (the number of hash functions) and m (the length of the bit vector B) increase. For better performance, we used the non-cryptographic `FNV` hash functions. To get better accuracy, $k = 16$ hash functions are used. Therefore, with $n = 256$, $m = 32n = 8192$ and $k = 16$, the false positive error rate is much less than 0.01.

The Bloom filter has been widely adopted for applications in other domains such as name lookup, spam detection and web caching. Wang et al. [12] implemented an approach for name look up in Named Data Networking called NameFilter which is a two-stage Bloom filter based scheme. Their experiments show that when the Bloom Filter is enabled, the memory consumption of their search scheme is reduced by 80% and the speed of name searching is 18 times faster than the traditional approach. Another benefit of using Bloom filters is reduced power usage. [11] proposed a low power Bloom filter architecture for network applications and the results showed that the architecture reduced the power consumption by 30%.

Therefore, we sought to improve the matching procedure by using a high speed and low computational resource consumption matching algorithm. Instead of matching each rule in a policy, or policy in a policy set, a highly efficient filtering mechanism can be added to filter out the policy fragments that *do not* satisfy the request, so that attention can focus on the remainder. Algorithm 1 shows how and where we integrate Bloom Filtering into the policy evaluation process. Note that applying the filter does not change the decision, because if the Bloom filter does not find a match, we are certain that no match exists. If it finds a possible match, we check anyway using the existing search algorithm.

Algorithm 1 EvaluatePolicyTarget(X, S)

Input: Index X of the *Policy* in a *PolicySet*. A set of *Attributes*(S) in the Request. The set of Bloom Filters (B) for the *PolicySet*, each B covers all Attributes in a given *Target*.
Output: A *Boolean* value to indicate if the policy is applicable to the request

```
p ← B[K] {assign the Bloom Filter for policy X to p}
for a ∈ S do
  positions ← hashk(a, k){hashk function computes the
  attribute a with number k of hash functions and returns
  a set of integers of positions at the Bloom Filter p}
  for i = 0 → k − 1 do
    if p[positions[i]]=0 then
      return false
    end if
  end for
end for
Continue on the standard target evaluation procedure
```

IV. Experimental Evaluation

A. Comparative PDP Selection

For our experimental evaluation, two PDP implementations were selected for the XACML 3.0 PDP comparative resource usage evaluation against **`Luas`**. The ATT-XACML and the Balana PDP implementations were selected as each complies with the XACML 3.0 standard and passed all conformance tests.

We also considered other PDP implementations that claimed high efficiency. XEngine has not been updated to support XACML 3.0 and supports only a very restricted subset of XACML 2.0, particularly "attribute = value" Target clauses and Rule Conditions only [6]. XACML 3.0 has been standardised since January 2013 and is widely deployed, so our performance experiment focuses only on implementations that are XACML 3.0 compliant. We considered SBA-XACML [5], but it does not provide the correct decisions for all valid XACML 3.0 policies and requests; we discounted sne-xacml [7] for the same reason. Therefore, we compared the only (open source) XACML 3.0 PDPs we could find that offer a complete implementation of the XACML 3.0 standard: ATT-XACML, Balana and **`Luas`**.

B. Evaluation Testbed

In order to learn the performance of PDP implementations in different circumstances, the experiment is designed to evaluate different high traffic workloads. Our testbed emulates real-world scenarios to evaluate PDP implementations. Each PDP is deployed in a Docker container. Each container is self-contained so the environment it presents to the user is specific to the deployment, but it uses the operating system level functions of the host and therefore is much lighter than a Virtual Machine that includes its own guest operating system. Docker provides a convenient platform for starting, stopping and for configuring containers and the applications therein.

System resources such as number of CPUs and the amount of memory allocated to each container can be easily controlled using the `dockerd` daemon. Since each container is segregated from the others, and dockerd starts each running instance with specific resources, it is easy to control the experimental conditions. The host is a commodity server with the following specifications: CPU has 4 cores, 16GB memory, Operating System is Ubuntu 18.04.1 LTS (64-bit) and Docker Version is 18.09.6. However, in order to emulate a resource-constrained

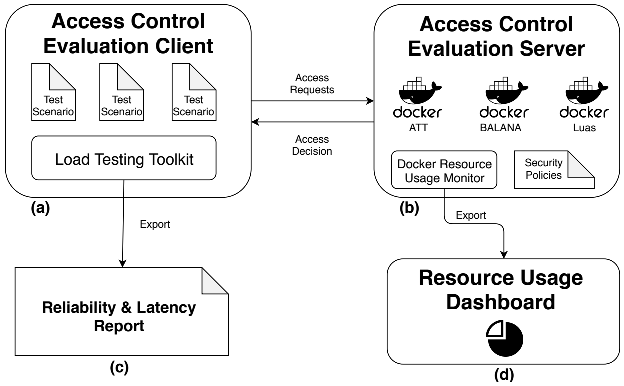

Fig. 1. System Diagram for XACML Evaluation Testbed. (a) is the access control client for sending access requests based based pre-defined test scenarios. (b) is the access control server which serves each PDP access control server via *Docker*. (c) is the aggregated report of reliability and latency, which is generated from (a). (d) is the dashboard to visualise matrices collected from (b)

device, we scale down the resources allocated to each container, so that it has 1 core with 512MB memory. The docker images built for the experiment are available on Docker Hub.

Figure 1 shows there are three major components in the testbed. The Access Control evaluation server runs the comparative PDP containers, each of which is pre-loaded with access requests. A docker resource usage monitor runs on the host and collects the real-time resource usage data for each running PDP container. The access control evaluation client reads the test scenario scripts for each run and tells the load testing toolkit to trigger a burst of requests in each container based on the pre-defined conditions. The triggered requests depend on evaluation parameters such as the number of active users and the number of requests that each user will send. The load testing toolkit we used for this evaluation is an open source application called artillery.io, which is able to simulate users or devices to send requests at high rates. This toolkit generates an evaluation report at the end of each test, which includes measurements such as request completion rates (relative to a given timeout limit), request processing time and failure rates. The resource usage data collected for each PDP container is exported to the resource usage dashboard where the data can be visualised. Each usage graph can be exported as an image or PDF, as desired.

C. Implementation and Validation

To increase its ability to process large numbers of requests, **`Luas`** follows the *Event-Driven* paradigm, so all the high I/O consumption features are designed to be asynchronous, otherwise the benefits of the underlying (asynchronous) event-driven approach are lost. For instance, the mechanism for reading policies uses an *asynchronous readable stream*. Comparing to the standard method to read a file, the *asynchronous readable stream* approach is more memory efficient and fast, because it reads the policy file one chunk at a time [13]. **`Luas`** also benefits from the modularization provided by JavaScript which improves its scalability and easy of use. **`Luas`** is packaged and released to NPM, which facilitates integration for developers.

TABLE I
PDP RELIABILITY: NUMBER OF REQUESTS COMPLETED VS NUMBER OF REQUESTS SENT FOR IOT

Sent Requests	AT&T	Balana	Luas
500	500 (100%)	500 (100%)	500 (100%)
1000	610 (61%)	1000 (100%)	1000 (100%)
1500	795 (53%)	1500 (100%)	1500 (100%)
2000	1020 (51%)	2000 (100%)	2000 (100%)

To ensure **`Luas`** makes correct decisions, per the published XACML standard [14], we used the XACML 3.0 Conformance Test suite [15] and ensured that **`Luas`** gave the right answers. However, we did not use this set of policies and requests for performance and resource testing. Instead, we used the well-known Continue set [16] and, to ensure that the responses are "correct", we checked that the access decisions of all 3 PDPs matched for all access requests, which they did.

D. Experiment Design

In order to understand the efficiency of each selected PDP, we designed four different scenarios to evaluate the PDP implementations to simulate real-world heavy load. Each PDP starts with the continue set policy, translated to XACML 3.0. This policy was chosen because it is widely used in PDP evaluation, e.g Liu et al. [4] used this set to test *XEngine*, Griffin et al. [17] also used the continue set for the same purpose. More recently, Morriset et al. [18] used continue as one of their test sets. The total number of rules in the continue set policy is 298. The load testing toolkit follows the scenario scripts we designed to simulate users sending access requests. In each scenario, 100 virtual users are simulated. In the first scenario, the load testing toolkit simulates 100 users issuing 5 requests per user; in the second scenario, 10 requests are sent from each user. Subsequent scenarios send 15 and 20 requests per user, respectively.

V. EXPERIMENTAL RESULTS

A. Reliability and Worst Scenario Analysis

The *reliability* metric is the ratio of the number of responses received by the client within a given time period after the time that request was issued by the client, relative to the number of requests sent by the client. For the experiment summarised in Table I, the requests are batched in bursts that are sent every second for $T_{\mathrm{send}} = 5$ seconds. This is done by creating $u = 20$ users per second, each of which sends $n/(uT_{\mathrm{send}})$ requests per second, where n is the total number of requests sent and T_{send} is the duration, in seconds, of the period when those requests are sent. Therefore, for this experiment, the number of requests per user per second is $n/100$.

We also note that reliability depends on a *response timeout*, which is set to $T_{\mathrm{response}} = 120$ seconds in all cases except AT&T with $n = \{1500, 2000\}$, when T_{response} is increased to 400 seconds, otherwise its reliability would be extremely low.

TABLE II
(95%, 99%) EVALUATION LATENCIES (IN MILLISECONDS SINCE ARRIVAL) PER PDP TYPE FOR IoT

Sent Requests	AT&T *(ms)*	Balana *(ms)*	Luas *(ms)*
500	(59201.7, 81051.1)	(8.2, 15.1)	(4.9, 8.6)
1000	(63351.2, 88927.3)	(10.1, 19.4)	(4.8, 7.9)
1500	(66105.9, 330922.3)	(69.7, 88.1)	(5.5, 9.1)
2000	(90780.7, 368400.5)	(133.1, 229.9)	(6.4, 10.6)

It is an open question whether $T_{\text{response}} = 400$ is acceptable in terms of quality of experience; even $T_{\text{response}} = 120$ is probably unacceptable for interactive applications.

Table I shows that the AT&T PDP implementation has the worst reliability compared to the other two PDPs. For example, even with a longer response timeout than its peers, it can process only 51% of the 2000 requests it received, because it spends so long on each evaluation that its extended response timeout of $T_{\text{response}} = 400$ seconds is exceeded. As seen by the client, 49% of the requests appear to have failed (in the sense that no response has arrived before the timeout). Therefore, we say that the AT&T PDP has a reliability score of 51% for this particular use case. The table also suggests that Balana and **`Luas`** are equally reliable because both succeed in providing responses to *all* arriving requests within the $T_{\text{response}} = 120$ seconds response timeout period.

Table II shows the request latency for each scenario in *percentiles* instead of the average. This is because users that are experiencing long access delays and those for whom access decisions are made quickly do not have the average experience. Anjum et al. [19] stated that using average as a performance indicator can be misleading since it is influenced by outliers, but performance *percentiles* provide a better sense for quality of experience (QoE).

Table II helps to explain the reliability scores. Even with 500 requests, the AT&T PDP has 95- and 99- percentiles of PDP service times that are approximately 60 and 80 seconds, respectively. Thus its worst case latencies are nearly 4 orders of magnitude greater than the equivalents for Balana and **`Luas`**. Hence it is not surprising that the AT&T reliability scores are not as good as those of Balana and **`Luas`**. However, it is also clear that **`Luas`** has more "performance headroom" than Balana, because its 95- and 99-percentiles are almost constant with respect to number of requests. By contrast, the percentiles for Balana grow super-linearly with the number of requests, indicating scalability problems ahead for Balana but not for **`Luas`**.

B. Resource Usage Analysis

In order to generate more accurate and precise results, `docker` is used to segregate each PDP run-time environment from its host. Each container includes a bare-minimum environment to serve the PDP via a web service. The docker image size of each PDP is: AT&T (668MB), Balana (660MB) and Luas (194MB). The bundle size of each PDP is: AT&T (329KB) Balana (485KB) and **`Luas`**(101KB). When deploying to resource-constrained devices at the network edge, **`Luas`** is more attractive than the other options because it requires less than a third of their (docker image) disk space.

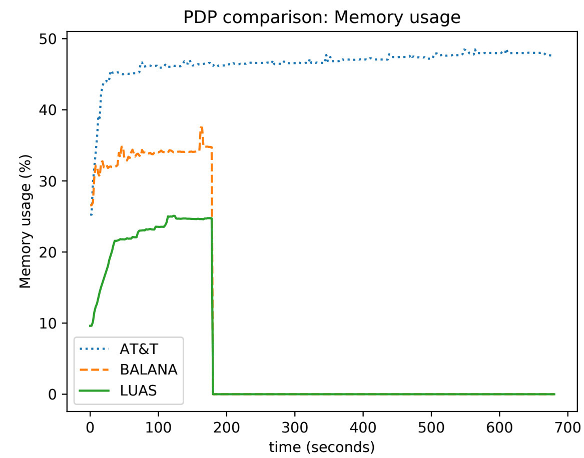

Fig. 2. Memory Usage % using data every 2 seconds from the Testbed Dashboard, for the 3 PDPs. **`Luas`** uses less memory during the experiment.

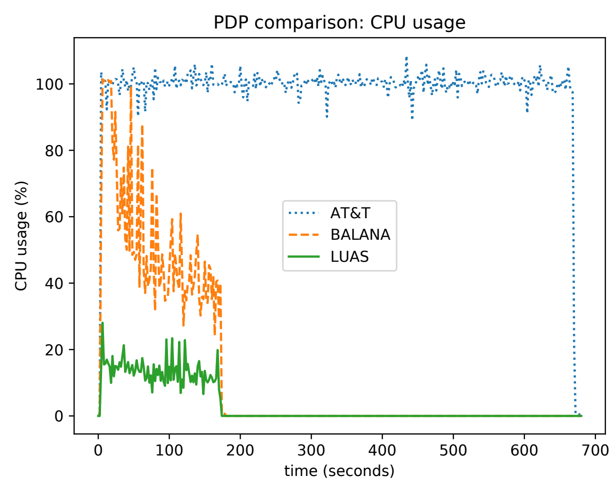

Fig. 3. CPU Usage % using data every 2 seconds from the Testbed Dashboard, for the 3 PDPs. **`Luas`** uses much less CPU during the experiment.

Figure 2 and Figure 3 were generated using data from the evaluation dashboard from the fourth experimental run, in which 2000 access requests were sent to each access control server. The independent axis represents time since the start of that run, so **`Luas`** and Balana finish their runs at approximately 180 seconds (3 minutes), which is why their profiles appear to be truncated relative to that of AT&T which is still processing requests more than 10 minutes after the experiment started.

Figure 2 indicates that **`Luas`** uses about 30% less memory than Balana and 50% less than AT&T when evaluating access requests against the same policy.

Figure 3 shows that the CPU consumption for **`Luas`** is dramatically less than that of the other two PDPs. This is partly because Node.js, its underlying runtime environment, is built with an event-driven model to provide a lightweight runtime environment. Greater CPU usage generally results in shorter battery life, which is a major consideration for devices at the network edge. Therefore, **Luas** is a more attractive candidate PDP for ubiquitous access control at the network edge than either of the other two PDPs.

These lower resource requirements (for **Luas**) are not achieved at the expense of evaluation accuracy, since all 3 PDPs in our comparison (eventually) return the same access decision when given the same access request.

VI. Conclusions

This paper evaluates the resource usage and performance of a new event-driven XACML implementation by comparing it against more traditional (blocking) implementations. From the above results it is clear that **Luas** achieves higher resource efficiency, better performance and greater reliability in a *resource-constrained environment*. Notably, **Luas** with its Bloom filter performs and scales better when it processes relatively high frequency requests sent from a large number of active users in contrast to other implementations using plain brute-force search, so it can help to solve the bottleneck in existing access control systems. Indeed, it manages to achieve greater performance and scalability while using fewer resources. Therefore, as a server component, it can be used as a drop-in replacement (offering the same API and responses) for Balana, say, while using fewer resources *and* offering higher performance.

The contributions identified in Section I were shown in Section III. **Luas** is open sourced and available via the Node Package Manager registry. **Luas** also has the advantage, relative to Balana, of using more modern web engineering and so has greater development potential.

These improvements are significant in practice, so it is now feasible to introduce robust security operations near the network edge, greatly enhancing the ability to manage security threats in domains where this was not considered possible before. There is no longer an excuse, in terms of additional latency, for not adding strong access controls to devices and operations in the Internet of Things.

We note that Bloom filters introduce trade-offs so we plan to investigate how best to configure them, balancing the Bloom filter's complexity against its accuracy to see how it affects performance and resource usage. We also wish to investigate how to integrate **Luas** into an *actual* device such as the MXE-100i Series IoT Gateway, since such devices offer a natural deployment target for **Luas**.

Acknowledgment

This work has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) and is co-funded under the European Regional Development Fund under Grant Number 13/RC/2077.

References

[1] H. Wei, J. Salvachua Rodriguez, and A. Tapiador, "Enhance OpenStack Access Control via Policy Enforcement Based on XACML," in *Proceedings of the 16th International Conference on Enterprise Information Systems - Volume 2*, ser. ICEIS 2014. Portugal: SCITEPRESS - Science and Technology Publications, Lda, 2014, pp. 283–289.

[2] B. Butler, B. Jennings, and D. Botvich, "An experimental testbed to predict the performance of XACML Policy Decision Points," in *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops*, May 2011, pp. 353–360.

[3] S. Jahid, C. A. Gunter, I. Hoque, and H. Okhravi, "MyABDAC: compiling XACML policies for attribute-based database access control," in *First ACM Conference on Data and Application Security and Privacy, CODASPY 2011, San Antonio, TX, USA, February 21-23, 2011, Proceedings*, 2011, pp. 97–108.

[4] 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, SIGMETRICS 2008, Annapolis, MD, USA, June 2-6, 2008*, 2008, pp. 265–276.

[5] A. Mourad and H. Jebbaoui, "Towards efficient evaluation of XACML policies," in *2014 Twelfth Annual International Conference on Privacy, Security and Trust*, July 2014, pp. 164–171.

[6] S. Pina Ros, M. Lischka, and F. Gómez Mármol, "Graph-based XACML Evaluation," in *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '12. New York, NY, USA: ACM, 2012, pp. 83–92.

[7] C. Ngo, M. X. Makkes, Y. Demchenko, and C. de Laat, "Multi-data-types interval decision diagrams for XACML evaluation engine," in *2013 Eleventh Annual Conference on Privacy, Security and Trust*, July 2013, pp. 257–266.

[8] B. Butler and B. Jennings, "Measurement and Prediction of Access Control Policy Evaluation Performance," *Network and Service Management, IEEE Transactions on*, vol. 12, no. 4, pp. 526–539, 2015.

[9] F. Turkmen and B. Crispo, "Performance evaluation of XACML PDP implementations," in *Proc. 2008 ACM workshop on Secure Web Services (SWS '08)*. ACM, 2008, pp. 37–44.

[10] R. R. McCune, "Node.js paradigms and benchmarks," 2011.

[11] M. Arun and A. Krishnan, "Multi Hashing Low Power Bloom Filter Architectures for Network Applications," in *2010 International Conference on Advances in Computer Engineering*, June 2010, pp. 1–5.

[12] Y. Wang, T. Pan, Z. Mi, H. Dai, X. Guo, T. Zhang, B. Liu, and Q. Dong, "NameFilter: Achieving fast name lookup with low memory cost via applying two-stage Bloom filters," in *2013 Proceedings IEEE INFOCOM*, April 2013, pp. 95–99.

[13] D. Grant, "fs.readFile vs streams to read text files in node.js," 2017, posted 22-August-2017.

[14] OASIS XACML-TC, "eXtensible Access Control Markup Language (XACML) Version 3.0," OASIS Standard, 2011.

[15] C. Dangerville, "XACML 3.0 Conformance Tests," 2017.

[16] S. Krishnamurthi, "The CONTINUE Server (or, How I Administered PADL 2002 and 2003)," in *Proc. Symposium on the Practical Aspects of Declarative Languages (PADL 03)*, ser. Lecture Notes in Computer Science 2562., Verónica Dahl and Philip Wadler, Ed. Springer, 2003, pp. 2–16.

[17] L. Griffin, K. Ryan, E. de Leastar, and D. Botvich, "Scaling Instant Messaging communication services: A comparison of blocking and non-blocking techniques," in *2011 IEEE Symposium on Computers and Communications (ISCC)*, June 2011, pp. 550–557.

[18] C. Morisset, T. A. C. Willemse, and N. Zannone, "Efficient Extended ABAC Evaluation," in *Proceedings of the 23Nd ACM on Symposium on Access Control Models and Technologies*, ser. SACMAT '18. New York, NY, USA: ACM, 2018, pp. 149–160.

[19] B. Anjum and H. Perros, "Adding Percentiles of Erlangian Distributions," *IEEE Communications Letters*, vol. 15, no. 3, pp. 346–348, March 2011.