Hardware-Enhanced Distributed Access Enforcement for Role-Based Access Control

Gedare Bloom
Dept. of Computer Science
George Washington University
gedare@gwu.edu

Rahul Simha
Dept. of Computer Science
George Washington University
simha@gwu.edu

ABSTRACT

The protection of information in enterprise and cloud platforms is growing more important and complex with increasing numbers of users who need to access resources with distinct permissions. Role-based access control (RBAC) eases administrative complexity for large-scale access control, while a client-server model can ease performance bottlenecks by distributing access enforcement across multiple servers that consult the centralized access decision policy server as needed. In this paper, we propose a new approach to access enforcement using an existing associative array hardware data structure (HWDS) to cache authorizations in a distributed system using RBAC. This HWDS approach uses hardware that has previous been demonstrated as useful for several application domains including access control, network packet routing, and generic comparison-based integer search algorithms. We reproduce experiments from prior work on distributed access enforcement for RBAC systems, and we design and conduct new experiments to evaluate HWDS-based access enforcement. Experimental data show the HWDS cuts session initiation time by about a third compared to existing solutions, while achieving similar performance to authorize access requests. These results suggest that distributed systems using RBAC could use HWDS-based access enforcement to increase session throughput or to decrease the number of access enforcement servers without losing performance.

Categories and Subject Descriptors

Security and privacy [**Security in hardware**]: Hardware security implementation

Keywords

access control, enforcement, hardware data structures

1. INTRODUCTION

Large enterprises have tens of thousands of employees who are distinct users that access tera- to peta-bytes of data composed of objects with tens to hundreds of thousands of distinct permissions [19]. Small businesses/enterprises have similar security needs as large businesses, but at a smaller and more flexible scale from tens to hundreds of users and hundreds to thousands of permissions. For large or small enterprises, data security is big business as intellectual property theft costs an estimated and growing 250 billion US dollars per year [22]. Hardening enterprises from external attacks is not sufficient, as over 85% of federal cases involving theft of trade secrets were perpetrated by insiders (former employee or business partner) [3]. Thus, protection mechanisms must control access to proprietary data by users in such a way that the users can execute their job functions without compromising the security of corporate intellectual property. The prevalent solution for enterprise data security is to use role-based access control (RBAC) [11, 24] with a client/server model. RBAC decouples users from permissions via roles, which are the principals to whom authorization is granted to access objects. A typical RBAC system will have about 3–4% as many roles as users [25], so an enterprise with 100,000 employees might use about 4000 roles. Although RBAC simplifies administration by reducing the number of principals, the number of permissions can cause performance bottlenecks in the software data structures used during access enforcement. We propose to enhance access control data structures with fast, parallel hardware to accelerate permission checking for RBAC systems.

In this paper, we enhance client/server RBAC with a hardware data structure (HWDS) that supports access control enforcement. HWDSs exploit hardware parallelism to reduce the asymptotic complexity of data structure operations, which can yield substantial performance improvements compared with software implementations [6]. A problem with the scale of enterprise security is that centralized access control becomes a bottleneck for the performance and availability of networked services, but centralization is viewed as necessary to facilitate policy administration. Therefore, modern systems divide access control into stages consisting of centralized authentication and a mix of centralized and distributed authorization and audit. (Audit happens off the critical path of access requests.) Authorization is split further into access decision and enforcement; the former consults policy to construct an authorization token, e.g. an access control list or capability, and the latter will grant/deny access to protected resources based on the token presented during an access request. Enforcement in this setting needs efficient mechanisms that can process authorizations without going to the centralized decision server; the solutions in prior work on client/server RBAC, which we review in Section 5, focus on caching, prefetching, and predicting authorizations.

Importantly, RBAC models use the notion of a *session* to permit users to activate subsets of their permitted roles when requesting an authorization, which supports good security practices such as the principle of least privilege [23] while making RBAC a good fit for client-server models such as the Common Open Policy Ser-

SACMAT'14, June 25–27, 2014, London, Ontario, Canada.

ACM 978-1-4503-2939-2/14/06 ...$15.00.
http://dx.doi.org/10.1145/2613087.2613096.

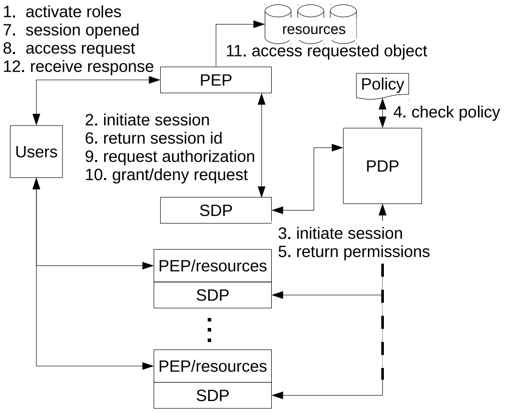


Figure 1: A client/server model based on the COPS standard extended with Secondary Decision Points.

vice (COPS) standard [14]. In typical client/server models, authorizations are created at the Policy Decision Point (PDP), which responds to Policy Enforcement Points (PEPs) that request whether or not to allow users to access protected resources. A common approach to improve PEP performance is to cache authorizations from the PDP so that repeated access requests may be enforced directly by the PEP without inducing communication and computation overhead to query the PDP. Crampton et al. [10] proposed adding a Secondary Decision Point (SDP) to cache and predict authorizations, which the authors called precise and approximate authorization recycling, respectively. Prediction (approximate recycling) infers authorizations for access requests that miss the cache. The architecture resulting from adding an SDP to the COPS model, shown in Figure 1, distributes authorization while maintaining a centralized policy. Wei et al. [28] show how to use the extended model with RBAC.

Good access enforcement performance requires an SDP with a high cache hit rate, no mis-predictions, and a low latency response time. In this paper, we demonstrate such an SDP using an associative array HWDS [5]. We implemented this HWDS-based SDP, which we call the HWDS BitSet SDP, using the gem5 [4] processor simulator and a Java library; we describe our implementation in Section 3. Our implementation enables performance evaluation of the HWDS BitSet SDP using the dist-rbac-eval benchmark [18, 2]. We improved the benchmark by adding skewed distributions of access requests and measurements of holistic SDP performance including session initiation and destruction times. The software library enables the SDP to use the pre-existing associative array HWDS without modification to the hardware, and the potential for commoditization together with good performance make the HWDS BitSet SDP well-suited to replace software SDPs in enterprise-scale RBAC deployments.

We evaluate the time efficiency of the HWDS BitSet SDP along with two SDPs—access matrix and CPOL—from prior work using the experiments described by Komlenovic et al. [18] to measure SDP access request time. The prior work evaluated SDPs using a series of experiments, which we reproduce to evaluate the new HWDS BitSet SDP. Furthermore, we describe and use new experiments to evaluate the effect on access request time due to the number and skewness in the distribution of access requests made during a session; see Section 4.

The experimental results show that the HWDS BitSet SDP typically is more efficient than the existing work for session initation and destruction, while remaining competitive for access request time. For workloads with 2 to 15 sessions having 100 roles, 250 permissions, 4 active roles per session, and each role has about 10 permissions, the session initiation time of the HWDS SDP has a speedup between 0.8 to 4.7 compared to the access matrix, and speedup of 1.46 to 2.61 over CPOL. Session destruction time for the HWDS SDP has speedup between 1.09 and 2.01 for access matrix, and 5.11 to 6.06 for CPOL. Averaged across these workloads, the time to initiate a session with the HWDS was 210.4 microseconds (μs) with standard deviation of 36, whereas the CPOL and access matrix SDPs took about 351.3 μs and 319.8 μs with standard deviations of 6.65 and 131.38, respectively. The averaged per-session destruction time was 30.3 μs for the HWDS with standard deviation 1.9, with CPOL at 165 μs and standard deviation 2.1, and the access matrix at 40.2 μs and 9.9 standard deviation. Note that some costs of SDP destruction go to garbage collection, which was not included in any of the reported data. The mean access time of the HWDS was between 0.15–0.6 μs, with CPOL taking 0.19–1.47 μs and the access matrix 0.15–0.49 μs. The results also indicate that when data are more skewed the HWDS BitSet SDP's access time is even more competitive.

The HWDS SDP can open and close sessions at a greater throughput than the other SDPs without losing access request time performance. A greater session throughput can be used to increase the number of active sessions each SDP can handle, which permits scaling up the number of users or scaling down the number of SDP servers. The advantage of a HWDS approach is that a HWDS can be used generically across multiple application domains in much the same way as software uses libraries containing optimized data structure implementations. The same HWDS that we use has also been used in part to improve SELinux performance [12] and network packet routing [21]. By integrating the HWDS support with software libraries, existing applications such as web browsers and physics simulators could benefit simply by linking to the modified library [5]. The HWDS approach is appealing because it improves data structure performance without requiring application- or problem domain-specific custom hardware designs.

This work makes three contributions. First, we show how to use a HWDS for distributed RBAC access enforcement. Second, we reproduce some of the experiments presented by Komlenovic et al. [18] and discuss how our findings compare with the previous results. Third, we describe new experiments and present results related to session timing results, skewness in the probability of access requests, and duration of sessions in terms of number of access requests made. The experimental results indicate the HWDS approach is a promising alternative to the best-performing software implementations of distributed access enforcement.

2. HWDS FOR DISTRIBUTED RBAC

The basic requirement of an SDP is to check access requests to determine whether a given session has the authorization (permissions) to access the requested resource. Associated with a session lifetime, the SDP must initialize and destroy session-specific data, i.e. the portion of the data structure related to the session. An SDP also must support updating its data structures to reflect any administrative changes that are made in the PDP's policy. These requirements motivate an associative array HWDS that stores a map between session identifiers and permissions as an SDP. An associative array, or map, is a data structure that organizes data to support efficient searching, or finding the node with a specific key from a set of (key, value) nodes. The specified key is the *argument* to the search [17]. Usual operations on an associative array are:

- *insert*: adds/replaces a key-value node

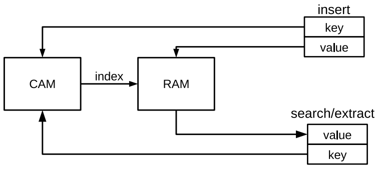


Figure 2: An associative array implemented in hardware. Keys are stored in the CAM, values are stored in the RAM.

- *extract*: removes a node
- *search*: finds a node using the given argument

An *exact* search returns the node with the same key as the argument if it exists. The *skewness* of a search is a measure of the asymmetry of the probability distribution of the argument. In this paper, we consider exact search with varying skewness.

The search operation supports access requests, and inserting a node with the same key as an existing node supports modifications to the permissions that a session can access in response to administrative changes. Thus, an associative array is a good candidate for an SDP implementation. An associative array can be implemented as a HWDS that supports numerical key-value pairs using a content-addressable memory (CAM) with a RAM of equal size, as shown in Figure 2. This HWDS has been used previously by others [21, 12, 5], which we discuss further in Section 5. CAMs are addressed by the stored data word instead of memory address, so given a data word, the CAM returns the memory address containing that word. To insert a key-value node, the map HWDS stores the key in the CAM and the value in the RAM at the same address in both memories. Extract and search operations use the argument to find the address of the value in the RAM and remove or return it respectively. The HWDS uses an extra 1 bit of RAM per data word as metadata to indicate whether that word is storing a value or garbage.

The advantages of a HWDS are parallelism and applicability to multiple application domains due to the generic hardware-software interface offered by data structure operations. Hardware can insert to a sorted structure faster than software because of the advantage of parallel comparisons. To insert a new node, a tree-based software data structure may need to traverse the depth of the tree, comparing the new node with the tree's node at each level. Hardware can compare the new node in parallel with all of the stored nodes. Similar performance improvements exist for other common data structure operations, such as deleting or searching for a node.

A HWDS reduces operation execution times to a small, fixed latency similar to a cache hit, but only for a data structure that has a maximum size less than HWDS capacity. Although software can request more storage for large data structures, a HWDS cannot request more capacity. The solution for a HWDS is to use runtime overflow management [5] so that arbitrary-sized data structures can utilize the HWDS resources. Overflow management consists of handling overflow and underflow conditions during HWDS operations. An overflow occurs when the HWDS has no more available space for an insertion. Underflow occurs when a HWDS operation misses in the hardware, for example an extract or search does not find the node in the HWDS.

2.1 HWDS BitSet SDP

We assume that a session identifier can be represented internally by an SDP as an opaque integer, and that a set of permissions map resources identified by integers between 0 and n inclusive to boolean values (true, false). The logical choice for a key-value pair for an SDP is to use the session identifier as a key, and some representation of permissions as a value. Although the map HWDS can implement an SDP by storing a pointer to a permission set in main memory as the value, the overhead of managing the HWDS and accessing main memory nullifies the performance advantage of the hardware. Another approach for using the map HWDS is to store a permission set directly in the RAM as a bitset (a vector of bits that represent the permissions of a session). This approach is problematic when the number of permissions is large, unknown a priori, or cannot be represented as a dense bitset.

Our approach, which we call *HWDS BitSet SDP*, is to create a session identifier and permission bitset for each session. The bitset is split into parts equal to the size of the data word that can be stored in the HWDS RAM. Then each part becomes a value, and the key is a combination of the session identifier and the offset of the part in the bitset. The key must be split between the session identifier and an offset into the permission bit vector. We choose to split the key in half, which is an arbitrary choice that users of the HWDS BitSet SDP can change by allocating more bits to the session identifier or the offset. The example in Section 2.2 uses an 8-bit key and value, which allows up to 2^7 permission identifiers (2^4 offsets, and 8 permission bits per offset) and up to 2^4 session identifiers. Our implementation uses a 64-bit value and 32-bit keys, so there can be 2^{22} permission identifiers (2^{16} offsets and 64 permission bits) and 2^{16} session identifiers.

When a session is initiated, all of the permissions for the active roles of the session are converted to a bitset, and then the bitset is divided into values. Each value is inserted to the HWDS with a key equal to the concatenation of the session identifier and the offset of the value in the bitset, padded with leading zeroes so the session identifier can be located correctly. The cost to initiate a session depends on how many values are created from the permissions, and is at most the number of offsets times the cost of a HWDS insert. However, when permissions exhibit good spatial locality in the bitset, the cost can be much lower, for example just one insert can represent up to 64 active permissions.

For an access request, the SDP computes the offset of the requested permission identifier by dividing the identifier by the size of the stored values. An argument for a HWDS search is constructed by concatenating the session identifier with the computed offset. The permission identifier's bit position is calculated by masking off bits in the identifier that are in positions greater than the size of the stored values. The bit at the calculated bit position is then checked in the return value from the HWDS search to determine whether the requested permission is in the session's permission bitset.

A session is destroyed by iterating over all possible offsets and extracting the node with a key equal to the session identifier concatenated with each offset. The cost to destroy a session can be quite large if the session has a large set of active permissions, so to reduce this cost the SDP tracks the maximum offset used in a session. During session destruction the number of HWDS extracts is equal to the maximum offset.

Overflow can occur when the number of HWDS inserts exceeds the capacity of the HWDS. The cost of overflow affects session initiation time to handle the overflow, and may affect access request or session destruction time to deal with associated underflows. Our implementation can hold up to 128 nodes before inducing overflow. A node permits one offset of 64 permissions per session, so without overflow a 128-node HWDS can hold 128 sessions with permissions ranging from 0–63, or 1 session with permission identifiers spanning from 0–8191. Overflow handling allows more sessions or permissions, but at some cost to move data from the HWDS to the overflow data structure.

Our approach to overflow management is *interposition-based* [5] similar to how Chandra and Sinnen [9] handle overflow for their priority queue HWDS. With interposition-based overflow management, HWDS operations are checked by software. If an operation causes an overflow, underflow, or any other problem, the software invokes a handler that corrects the problem and either emulates or replays the operation.

Overflow can be governed by different policies about which nodes to move between the HWDS and overflow storage. The associative array HWDS we use supports three policies for spilling overflow nodes from the hardware: least-recently used (LRU), least-frequently used, and priority-ranked by integer comparison of keys. The HWDS BitSet SDP's overflow handler moves half of the nodes from the HWDS to a software implementation of an associative array using the LRU policy. For underflow, the handler passes through the operation to the software associative array: for a search, if the node is found in the software data structure, that node is removed and re-inserted to the HWDS. The policies governing overflow and underflow were chosen to exploit temporal locality.

Discussion.

We considered several alternatives before settling on the design of the HWDS BitSet SDP. The primary consideration for this design was to avoid introducing custom hardware modifications to the existing HWDS logic. We considered loading each permission separately into the HWDS, but sessions with a lot of permissions would put pressure on the HWDS capacity while inducing high session initiation and destruction costs. We also considered loading a single pointer reference to a permission bit-vector in the HWDS, but we found the overhead to fetch the bit-vector from memory and then compute the permission offset and mask led to poor performance in small or sparse sets of permissions. Loading the HWDS with chunks of the bit vector offers a good compromise between search speed and HWDS utilization. An alternative to interposition-based overflow management is exception-based, which relies on the hardware to check for error conditions and raise an exception in case of a problem. We found that the cost of passing an exception from hardware through the operating system into the JVM and finally to the Java application was prohibitive, so we used interposition exclusively.

2.2 RBAC Example

Figure 3 depicts an RBAC policy and an example of HWDS BitSet SDP operations. Although our implementation uses 64-bit values and 32-bits keys, this example uses 8-bit values and keys for clarity.

Suppose Alice wants to approve a time card and enter her time card. She will start by initiating a session activating the roles of Manager and Employee. The PEP will forward this request through the SDP to the PDP, which responds with a new session identifier, say 3, and the set of permissions for the role Manager, which is {12, 6, 10}. The HWDS BitSet SDP will use the returned session identifier and permissions as follows. First, the set of permissions is converted to a vector of bits with a set bit at each position of the allowed permissions: 0001 0100 0100 0000. This bit vector is then split into multiple values, 0001 0100 and 0100 0000 corresponding to the two 8-bit values that compose the bit vector. The offset of the values are 0 and 1, respectively, and recall that the session identifier is 4, so the keys are 1000 and 1001. Thus, the SDP inserts two nodes into the HWDS: insert(1000, 01000000) and insert(1001, 00010100). The SDP also returns the session identifier (4) to the PEP, which forwards the identifier to the user.

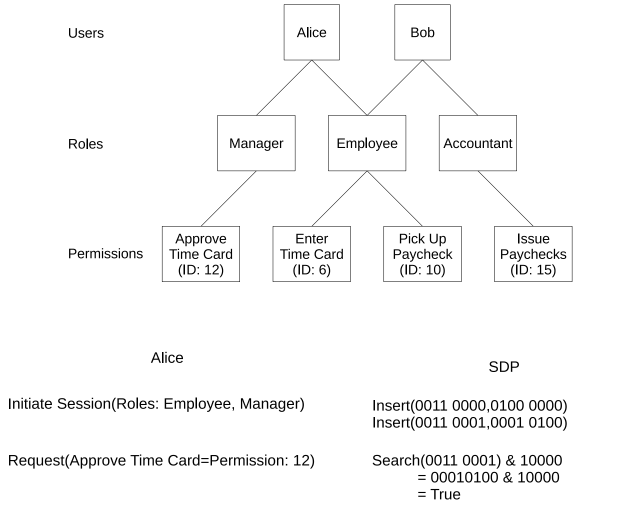

Users
Alice
Bob
Roles
Manager
Employee
Accountant
Permissions
Approve Time Card (ID: 12)
Enter Time Card (ID: 6)
Pick Up Paycheck (ID: 10)
Issue Paychecks (ID: 15)
Alice
SDP
Initiate Session(Roles: Employee, Manager)
Insert(0011 0000,0100 0000)
Insert(0011 0001,0001 0100)
Request(Approve Time Card=Permission: 12)
Search(0011 0001) & 10000
= 00010100 & 10000
= True

Figure 3: An example RBAC policy and a session initiation and access request using the HWDS BitSet SDP. Note that the session identifier is 3, and the key is split into 4 bits for the session identifier and 4 bits for the offset in the permission bit-vector.

Now that Alice has opened a session, she can approve a time card. She issues a request to the PEP to access the resource associated with approving time cards. The PEP translates her request into an access request to the SDP for session identifier 4 and permission 12. The SDP calculates the offset of the requested permission by dividing 12 / 8 and discarding the remainder, getting an offset of 1. The SDP then computes the argument by shifting the session identifier and merging it with the offset, that is the SDP executes search(1001) and gets back the 8-bit value (00010100) associated with offset 1. The SDP then computes the bit position of permission 12 in that value by masking off $(12 \,\&\, 7)$, obtaining 4. (If instead Alice had requested to enter her time card—permission 6—the offset would be 0 and the bit position of the permission in the returned value would be 6.) Finally, the SDP checks if the returned value has a set bit at position 4, which it does, so the SDP responds to the PEP that the access should be granted. The PEP allows Alice to approve time cards.

3. EXPERIMENTAL SETUP

We implemented the associative array HWDS in the gem5 simulator [4]. Our implementation introduces a new gem5op—a pseudo instruction that executes atomically and non-speculatively—that software executes to perform a HWDS operation. We wrote a software library that encapsulates HWDS operations in a Java class using the Java Native Interface to execute gem5ops. The library and the underlying HWDS use 32-bit keys and 64-bit values, and can model arbitrary-sized HWDSs; we use only a 128-node HWDS, i.e. 128 key-value pairs can be stored in the hardware at a time. The library accepts the size parameter in its class constructor, which programs the size into the HWDS during its initialization. Every HWDS operation returns a value in a register that the library checks for error codes that signal the library to handle the problems of overflow and underflow as described in Section 2.1. The library uses a Java Collections TreeMap as the software associative array for overflow.

3.1 Benchmark

We adapted the dist-rbac-eval benchmark for RBAC access enforcement described by Komlenovic et al. [18], which the authors made available online [2]. This benchmark evaluates the time-efficiency of SDP-based access enforcement for RBAC. An RBAC policy is encoded as a directed graph by the benchmark, which calls a specific instance of a policy an RBAC configuration. For each configuration, multiple session profiles can be created, each of which is a sequence of instructions comprising session initations, access requests, and session destructions. The benchmark executes the session profile over the RBAC configuration for an SDP implementation, six of which are provided from the prior work by Komlenovic et al. [18]: access matrix, Bloom filter, cascade Bloom, CPOL, directed graph, and authorization recycling.

3.2 Session Profile Modifications

We introduce a new parameter for session profiles that controls the access request distribution. The default behavior for the unmodified benchmark is to choose a session at random and generate sequentially-ordered access requests for permissions available to that session. Access requests are generated until the limit on the number of access checks is reached. If all of the permissions for a session have been requested, a new session is activated and access requests are issued until the limit is reached. We introduce a new parameter α to the class such that when $\alpha = 0$ the permissions used in access requests are chosen uniformly at random, and when $\alpha = 1.0$ the permissions follow Zipf's law [17].

Zipf's law, or Zipf's distribution, is a power law distribution that, applied to access requests, means the probability of the nth most common resource (permission) is requested with probability approximately equal to $1/n$. This distribution originated from observations made about the popularity of words in languages, and has been observed in web document popularity [8]. When using the new parameter, access requests are made at random according to the skewed distribution across all active sessions and permissions until the limit on access requests is reached. The reason to introduce the new parameter α is that some data structures optimize for skewed search patterns, for example the splay tree is a self-adjusting binary tree that moves frequently-accessed nodes closer to the root. Indeed, many cache-like structures will do best on data that are skewed. Thus, we aim to quantify whether some SDPs may do better when access requests are skewed.

3.3 Benchmark Modifications

We modified the benchmark to reduce its execution time so that we could simulate realistic workloads in a timely manner. Importantly, the benchmark was changed to instantiate only one PDP and to pre-compute the PDP's responses by running through one iteration of the benchmark's workload before beginning the iterations that time the SDP. These changes remove the overhead of PDP instantiation and policy evaluation from the benchmark's workload loop, which substantially reduced the benchmark runtime without sacrificing accurate timing of SDP access enforcement; we validate the timing is correct by reproducing experiments conducted in the prior work. We estimate these changes resulted in a speedup of about 7–10 in the time to run the benchmark. The time for one run of the benchmark in gem5 for the three SDPs we used was reduced from about a day to just over 3 hours, which allows running the benchmark multiple times with different configurations and session profiles as needed by the experiments we conducted. Modifications to the PDP required rewriting the initiation code for the SDPs. Since the access matrix and CPOL SDPs were the most time-efficient in the prior work, we chose to focus on those two SDPs along with our implementation of the HWDS BitSet SDP.

The focus in the literature on mean access time as a proxy for SDP performance is misleading in case an SDP has extremely poor session initiation and destruction times. Many data structures can optimize the search path at the expense of the insert/delete path, for example a balanced search tree does extra work to keep the tree height in balance during insertions and deletions so that searches achieve logarithmic asymptotic time complexity. The time it takes to start a session can negatively impact a user's experience especially if the delay is noticeable. Therefore, we aim to quantify the effect of the SDP on session initiation and destruction. To do so, we added timing measurements to the benchmark for session initiation and destruction that measure the time needed by the PDP to construct and destroy session authorizations for the SDP. By measuring the initiation and destruction costs for the SDP data structure, the benchmark now estimates the entire cost of a particular SDP implementation.

3.4 SDP Implementation

We implemented the HWDS BitSet SDP in the dist-rbac-eval benchmark using our gem5 implementation of the associative array HWDS. The HWDS BitSet SDP works as described in Section 2. A list of permissions associated with a session is converted into a Java BitSet by the PDP, which returns the BitSet and session identifier to the SDP. The SDP extracts and inserts each 64-bit chunk of the BitSet into the HWDS with a key constructed from the session identifier and offset of the chunk. An access request computes the offset for a requested permission and searches the HWDS using an argument constructed from the session identifier and offset. The return value is checked at the bit position of the requested permission and access is granted (denied) if the bit is set (clear). A session is destroyed by extracting keys for every offset for the session.

3.5 Methodology

The hardware platform is a simulated system using the gem5 simulator with the timing CPU, which is a simplified processor core that models the timing of memory accesses and cache, but it does not include detailed pipeline interactions or out-of-order execution. For the gem5 platform, we use the X86 full system simulation with 1 core and 512 MB of RAM, 64K data cache, 32K instruction cache, running at 500 MHz, with Linux 2.6.28.4, java version 1.7.0_51, and the OpenJDK VM. We set the VM memory heap size to 512 MB. We use this platform to evaluate the HWDS BitSet SDP.

Our methodology for executing the benchmark is similar to that of Komlenovic et al [18], but we did make some minor variations. This methodology is inspired by the work of Georges et al. [13], who present a detailed analysis of Java performance analysis and experimental design with Java. Most important, Java programs exhibit two primary phases with distinct performance characteristics, a startup phase and a steady-state phase. Experiments that measure performance in Java must take into consideration which phase to observe. We are interested in the steady-state performance of the SDP, therefore we use the four-step methodology proposed by Georges et al [13]. However, we do not run the VM multiple times because executing the benchmark on the same inputs and parameters gives the same timing results in the simulator, so there is no variance between VM invocations. Each invocation runs at most 25 iterations of the session profile for the benchmark. The first 16 iterations are ignored, and the benchmark terminates when 5 iterations in a row achieve a mean access time with a coefficient of variation (CoV) less than 0.02. (CoV is equal to the sample standard deviation divided by the sample mean.) If the CoV of 0.02 is not achieved within 25 iterations, the benchmark terminates and reports the 5 consecutive iterations with the smallest CoV. We record the access time along with the session initiation and destruction times for these 5 iterations for each VM invocation. The first 16 iterations are ignored because we found that the variation between iterations is small even in the startup phase, which causes a small CoV that terminates the benchmark before it reaches the steady-state.

In the experimental results, we report mean access time performance that was calculated as the mean of the recorded access times across the repeated VM invocations. We also compute and report the 95% confidence interval for each mean. For the initiation and destruction times, we calculate the average of the times reported across all VM invocation, and we discard any measurement that is outside of 1.5 times the interquartile range. The reason to discard these outliers is that the initiation and destruction times in the steady-state are still affected by Java-induced variations, especially garbage collection. Discarding the outliers gives a more accurate estimate for the performance of session initiation and destruction without garbage collection.

4. EXPERIMENTS AND RESULTS

We conducted a series of experiments using the experimental setup described in Section 3.

4.1 Reproduced Experiments

The first set of experiments aim to reproduce those presented by Komlenovic et al. [18] using the benchmark. We reproduce the prior work in order to validate the modifications we made to session profiles and the benchmark workload, and also to determine useful parameters for executing the benchmark on the gem5 platform. We chose the parameters used in these reproduced experiments to match the parameters of the prior work.

We attempted to recreate the RBAC configurations and session profiles for the inter- and intra-session experiments described in the prior work. An inter-session experiment consists of multiple benchmark executions with increasing numbers of sessions for each execution. For the inter-session experiment configurations, we set the number of users to 25, roles to 100, permissions to 250, roles per user to 4, and roles per permission to 3. The benchmark is agnostic to the number of users, since each user can generate multiple sessions in parallel by activating different sets of roles and permissions. 100 roles is reasonable for an organization with approximately 2500 users considering 4% ratio of users to roles.

An intra-session experiment executes the benchmark multiple times for the same number of sessions and fixed number of permissions with varying roles, and fixed number of roles with varying permissions. For the intra-session experiment configurations, we set the number of sessions to 15, users to 2, and varied the roles and permissions.

The inter-session experiments investigate SDP performance as parameters related to multiple sessions changes, whereas the intra-session experiments investigate SDP performance with fixed session parameters and varying other RBAC configuration and session parameters that can affect performance.

The results from running the reproduced experiments for the access matrix and CPOL SDPs are presented in the appendix. Using the same configurations and session profiles as in the reproduced experiments, we also measured session initiation and destruction times as described in Section 3.5. The results for session initiation and destruction times are in the appendix.

To evaluate the effect of skewness on SDP performance, we generated session profiles with the parameter α equal to 0, 0.25, 0.5, 0.75, and 1.0. We found no statistically significant difference in mean access time between the access matrix and CPOL SDPs with increasing access request skewness; that is, the mean confidence intervals overlap even for the extreme cases of $\alpha = 0.0$ and $\alpha = 1.0$. We attribute this lack of difference to the benchmark's SDP being sufficiently small enough to fit into cache that the access pattern has no effect on performance since these two SDPs both prefetch the entire permissions data needed for authorizations.

4.2 HWDS BitSet SDP

To evaluate the effectiveness of the HWDS BitSet SDP, we conducted similar experiments to those described above. For these experiments, we focused on the inter-session behavior of the HWDS BitSet SDP, and the intra-session behavior for the increasing number of permissions.

As the number of sessions and permissions increase, the expectation is that the HWDS BitSet SDP performance will degrade due to overflow handling. In particular, the threshold for overflowing a HWDS is when the number of inserted nodes exceeds the number of sessions times the maximum permission identifier divided by 64. So with 250 permissions, overflow occurs when there are more than 32 sessions. We therefore varied the number of sessions up to 60 to observe the effect of overflow. 60 sessions requires approximately double the capacity of the HWDS. Each session generates 1000 access requests and we used α equal to 0.0 and 1.0.

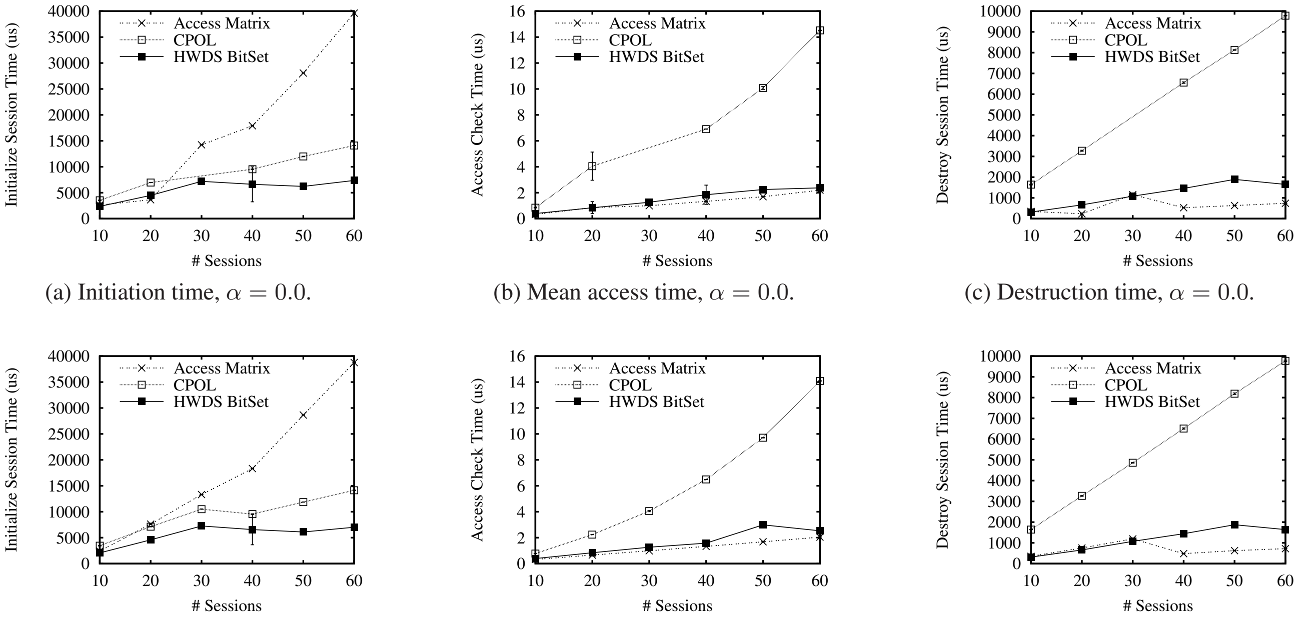
We executed the benchmark using the inter-session profiles and access matrix, CPOL, and HWDS BitSet SDPs in the gem5 simulator. Each session generates 1000 access requests and we investigated both α equal to 0.0 and 1.0. Figure 4 shows the results for these benchmark executions. The total session initiation time averaged across the steady-state iterations used to compute the mean access time is presented in Figure 4a for $\alpha = 0.0$ and in Figure 4d for $\alpha = 1.0$. Figures 4.2 and 4.2 depict the mean access time for $\alpha = 0.0$ and $\alpha = 1.0$ respectively. and Figures 4c and 4f shows the session destruction time.

The mean access time and session destruction time for the HWDS BitSet SDP and the access matrix track closely, and they both outperform CPOL. Session initiation time with the HWDS BitSet SDP is about one-third less than with CPOL, and as the sessions increase the problems with access matrix session initiation are evidenced by the divergence between the HWDS BitSet and access matrix initiation times. With 15 sessions, the HWDS BitSet session initiation time is 28% less than the access matrix SDP's time, and the gap widens as the number of sessions increases. The per-session destruction time in the inter-session experiments was between 34.6–64.2 μs for the access matrix, 27–32.1 μs for the HWDS, and 163–170 μs for CPOL.

Even in the presence of HWDS overflow, the performance of the HWDS BitSet SDP is better than the software SDPs. Overflow occurs in the inter-session profiles when the number of sessions exceeds 32, and in the intra-session profiles when the number of permissions exceeds 546. When there is overflow, the HWDS BitSet SDP performs better with skewed access requests (Figure 4.2) than with uniform access requests (Figure 4.2), because the LRU overflow handling exploits temporal locality.

In the reproduced experiments, we found that the access matrix suffers heavy performance degradation for session initiation with the intra-session profiles, and it especially scales poorly with the increasing numbers of permissions. Thus, we executed the intra-session profiles using only the CPOL and HWDS BitSet SDPs in the gem5 simulator. Neither of these SDPs are affected by changing the number of roles, so the intra-session profiles use 100 roles and take the number of permissions from {100, 500, 700}. With 700 permissions, 15 sessions, and 128-node capacity, the HWDS BitSet SDP overflows 37 nodes.

Figure 5 shows the mean access time, average total session initiation time, and average total session destruction time as the number



(a) Initiation time, $\alpha = 0.0$.

(b) Mean access time, $\alpha = 0.0$.

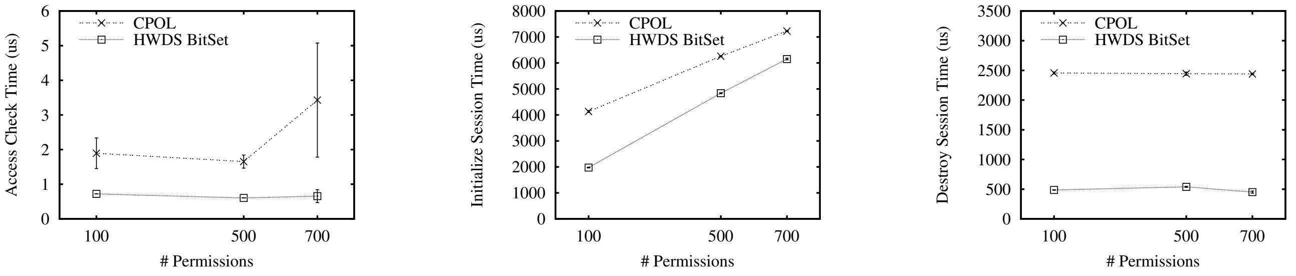
(c) Destruction time, $\alpha = 0.0$.

(d) Initiation time, $\alpha = 1.0$.

(e) Mean access time, $\alpha = 1.0$.

(f) Destruction time, $\alpha = 1.0$.

Figure 4: Inter-session results



(a) Mean access time

(b) Session initiation time

(c) Session destruction time

Figure 5: Intra-session results for varying permissions using gem5.

of permissions changes. For these experiments, we used 1000 access requests per session and set α equal to 0.0. Figure 5a shows that the HWDS BitSet outperforms CPOL in terms of access check time. The mean access time for these data exhibit large variance for CPOL, none of which resulted with a CoV less than 0.02, though all of the HWDS BitSet measurements do. Such large variance indicates the experiment does not reach a stable steady-state when using the CPOL SDP. The session initiation time of the HWDS BitSet SDP scales worse as the number of permissions increases, since more permissions causes increased overflow while constructing the SDP data structure. Figure 5 shows how the HWDS BitSet SDP compares with the CPOL SDP for initiation time when scaling the number of permissions; the improvement is 52% at 100 permissions but only about 15% at 700 permissions.

The experimental results suggest that the HWDS BitSet SDP is a good choice for a high-performing SDP. We found that the HWDS BitSet performs comparably to the best SDP implementation for mean access time and session destruction, and the HWDS BitSet outperforms the prior work for session initiation time in all of the experiments, even those which induce overflow handling.

5. RELATED WORK

Access control solutions in commercial distributed systems—for example, IBM Tivoli [15], Oracle Entitlements Server [1], and CA SiteMinder Web Access Manager [16]—replicate policy to improve access enforcement performance when authorization is divided. Replication reduces the bottleneck of a centralized policy, but at the increased cost to keep the replicas consistent. Authorization still incurs computation time overhead to query the policy database, which others have argued can be the limiting factor in access request throughput [7]. Furthermore, communication overhead will degrade authorization performance unless each enforcement point has its own replica.

Policy replication is complementary to other approaches that improve the time efficiency of access enforcement in distributed systems, namely caching, predicting, and prefetching. Caching authorizations is a well-known and widely-deployed mechanism to improve the performance of access enforcement [26, 16, 1]. Authorization caching works well when access requests are repetitive and exhibit good temporal locality. In case access requests are not repeated exactly, Crampton et al. [10] proposed the secondary and approximate authorization model (SAAM) introducing approximate authorization recycling. Approximate recycling predicts an authorization without consulting the PDP during an access request. Wei et al. [28, 29] applied the SAAM to RBAC. Good performance for authorization recycling depends heavily on cache warmness, which can be a problem especially for short duration sessions or access requests with low temporal locality. To circumvent the problem of a cold cache, Tripunitara and Carbunar [27] proposed implementing an SDP using a *push* model that prefetches authorizations into the SDP. The authors proposed the novel cascade Bloom filter as a data structure for caching authorizations. Komlenovic et al. [18] proposed and implemented a Java benchmark to evaluate implementations of 6 SDPs for RBAC. The 6 SDPs included reproductions of the above SAAM authorization recycling and cascade Bloom filter, along with new implementations using a directed graph representation, access matrix encoding derived from the work of Liu et al [20], Bloom filter, and a re-implementation of CPOL [7]. Other than authorization recycling, these SDPs use the same push model as Tripunitara and Carbunar. The HWDS BitSet SDP adopts the push model to provide a prefetched authorization cache at the SDP.

The HWDS BitSet SDP also extends some of the prior work in the area of hardware support for access control and hardware support for data structures. Our implementation of the associative array HWDS is inspired by the map HWDS used by Bloom [5] for generic search and Fiorin et al [12]—see below. The Java interface and implementation of overflow handling we use is similar to that used by Chandra and Sinnen [9] for their priority queue HWDS. The novelty of our work, however, is not in the design of a HWDS or its overflow handling, but rather in applying and evaluating the HWDS to the domain of RBAC and interfacing the HWDS with the Java environment of the access enforcement benchmark.

Hardware support for access control is not new—indeed, most modern systems use CAM hardware with memory protection bits to implement the translation lookaside buffer, which does virtual to physical memory address translation. Even hardware support for RBAC has been proposed by Fiorin et al. for improving the performance of SELinux by replacing the access vector cache with a similar hardware design as the associative array HWDS [12]. Our work differs from the prior work on hardware-enhanced access control by using only the HWDS as the hardware component, and the rest of the access enforcement is implemented in software.

6. CONCLUSION

In this paper, we presented a novel SDP, the HWDS BitSet SDP, that uses an associative array HWDS to store permissions in bit vectors. We evaluated this SDP using an open-source benchmark for distributed access enforcement, and we found that the time-efficiency of the HWDS BitSet SDP is competitive with the best SDP implementations provided with the benchmark. By including the cost of session initiation time in the benchmark's measurements we also found that the HWDS SDP takes much less time—about one-third less—to instantiate the data structure for caching session authorizations. The experiments we conducted included reproductions of a prior study [18] using the benchmark, and new experiments we designed that investigate the effect on access request times due to skewed probability distributions in access request patterns and to duration of sessions in terms of numbers of access requests. The experimental results for the reproduced experiments are commensurate to those reported by the prior work. We found that skewness in access requests does not affect mean access time for the access matrix and CPOL SDPs. However, the number of access requests made in a session has a strong influence on SDP performance—especially of CPOL—with larger access times observed for shorter sessions, which we attribute to not reaching the best steady-state phase fast enough. These experimental results suggest that the HWDS BitSet SDP is a promising approach to improve the performance of distributed access enforcement in a client-server RBAC system.

While this paper makes a step in enhancing access control with hardware support, future work can further investigate how the HWDS approach can be used to improve access enforcement. Some areas that remain open to explore include the hardware design space, new HWDS operations that can support more efficient access enforcement, and applying the HWDS approach to other access control domains such as in single-user systems. In terms of the hardware design, the CAM-based HWDS is power-hungry and does not scale well beyond the 128-entry size we modeled in this paper. Alternative HWDS designs could scale better to larger capacities without sacrificing the performance advantages of hardware parallelism. Furthermore, instructions could be added to the HWDS that would benefit its use for access enforcement: an operation that can extract all of the nodes with the same session by matching a subset, or mask, of the key would reduce the cost of session destruction to the time needed to execute a single HWDS operation; operations to audit the use of the HWDS would benefit practical deployments by making accountability part of using the hardware; and operations that divide an array into multiple key-value nodes automatically would reduce some of the costs associated with loading permission identifiers into the HWDS.

7. ACKNOWLEDGMENTS

The authors are supported in part by NSF CNS-0934725. The authors thank the reviewers and shepherd, Mahesh Tripunitara, for helping to improve the presentation of our work. The modified benchmark source code and experimental harness we used is available at https://github.com/gedare/dist-rbac-eval/tree/SACMAT14

8. REFERENCES

[1] Fine grained authorization: Technical insights for using oracle entitlements server. Technical report, Oracle, 2012.

[2] dist-rbac-eval - a platform for assessing approaches to distributed role-based access control (RBAC) enforcement https://code.google.com/p/dist-rbac-eval/, 2014.

[3] D. S. Almeling, D. W. Snyder, M. Sapoznikow, W. E. McCollum, and J. Weader. A statistical analysis of trade secret litigation in federal courts. *Gonzaga Law Review*, 45(2):291–334, 2010.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[5] G. Bloom. *Operating System Support for Shared Hardware Data Structures*. PhD thesis, The George Washington University, Jan. 2013.

[6] G. Bloom, G. Parmer, B. Narahari, and R. Simha. Shared hardware data structures for hard real-time systems. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '12, pages 133–142, Tampere, Finland, 2012. ACM.

[7] K. Borders, X. Zhao, and A. Prakash. CPOL: high-performance policy evaluation. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 147–157, Alexandria, VA, 2005.

[8] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. In *IEEE INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings*, pages 126–134, New York, NY, Mar. 1999.

[9] R. Chandra and O. Sinnen. Improving application performance with hardware data structures. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–4, Atlanta, GA, USA, Apr. 2010.

[10] J. Crampton, W. Leung, and K. Beznosov. The secondary and approximate authorization model and its application to bell-LaPadula policies. In *Proceedings of the Eleventh ACM Symposium on Access Control Models and Technologies*, SACMAT '06, pages 111–120, Lake Tahoe, CA, 2006.

[11] D. F. Ferraiolo and D. R. Kuhn. Role-based access controls. pages 554–563, Baltimore, MD, Oct. 1992.

[12] L. Fiorin, A. Ferrante, K. Padarnitsas, and F. Regazzoni. Security enhanced linux on embedded systems: A hardware-accelerated implementation. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 29–34, Sydney, NSW, 2012.

[13] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, Montreal, Quebec, Canada, 2007. ACM.

[14] Jim Boyle, Ron Cohen, David Durham, Raju Rajan, Shai Herzog, and Arun Sastry. The COPS (common open policy service) protocol. Technical Report 2748, IETF, Jan. 2000.

[15] G. Karjoth. Access control with IBM tivoli access manager. *ACM Trans. Inf. Syst. Secur.*, 6(2):232–257, May 2003.

[16] Kire Terzievski, Steven Turvey, and Matt Tett. CA WAM solution hundred million user test. Technical Report 080202, Enex TestLab, Jan. 2009.

[17] D. E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., 1998.

[18] M. Komlenovic, M. Tripunitara, and T. Zitouni. An empirical assessment of approaches to distributed enforcement in role-based access control (RBAC). In *Proceedings of the first ACM conference on Data and application security and privacy*, CODASPY '11, pages 121–132, San Antonio, TX, USA, 2011. ACM.

[19] Laura DuBois and Natalya Yezhkova. Distinctions between SMB and enterprise requirements for protection, archiving, and recovery. Technical report, IDC, Framingham, MA, USA, Apr. 2009.

[20] Y. A. Liu, C. Wang, M. Gorbovitski, T. Rothamel, Y. Cheng, Y. Zhao, and J. Zhang. Core role-based access control: Efficient implementations by transformations. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '06, pages 112–120, Charleston, South Carolina, USA, 2006. ACM.

[21] K. Pagiamtzis and A. Sheikholeslami. Content-addressable memory (CAM) circuits and architectures: a tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3):712–727, Mar. 2006.

[22] Preeta M. Banerjee and Eric Openshaw. Democratizing technology: Crossing the "CASM" to serve small and medium businesses. *Deloitte Review*, (14), Jan. 2014.

[23] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[24] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[25] A. Schaad, J. Moffett, and J. Jacob. The role-based access control system of a european bank: A case study and discussion. In *Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies*, SACMAT '01, pages 3–9, Chantilly, Virginia, USA, 2001. ACM.

[26] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The flask security architecture: System support for diverse security policies. In *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*, SSYM'99, pages 123–139, Washington, D.C., USA, 1999. USENIX Association.

[27] M. V. Tripunitara and B. Carbunar. Efficient access enforcement in distributed role-based access control (RBAC) deployments. In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*, SACMAT '09, pages 155–164, Stresa, Italy, 2009. ACM.

[28] Q. Wei, J. Crampton, K. Beznosov, and M. Ripeanu. Authorization recycling in RBAC systems. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 63–72, Estes Park, CO, 2008. ACM.

[29] Q. Wei, J. Crampton, K. Beznosov, and M. Ripeanu. Authorization recycling in hierarchical RBAC systems. *ACM Trans. Inf. Syst. Secur.*, 14(1):3:1–3:29, June 2011.

APPENDIX

A. EXTENDED RESULTS

The experimental results presented here do not use the HWDS BitSet SDP, so the experiments are conducted using a typical user workstation. The hardware platform for these experiments was a 2.5GHz Intel Core2 Quad (Q9300) with 3 GB of main memory running CentOS 6 with Linux 2.6.32, java version 1.7.0_51, and the OpenJDK VM; we set the maximum memory usage for the VM to 1.5 GB. (Note: the benchmark is single-threaded, so only one core was used. The system was otherwise idle.) We use this platform to validate that the performance of the software-only SDPs are commensurate with the prior work, and to explore parameters for the new experiments measuring session initation and destruction times, and the effect of skewed access requests.

With this platform, we execute the VM 4 times, and ignore the first 8 iterations of the each invocation. If the CoV of a measured mean access time is larger than 0.02, then we discard that measurement for the Intel Core2 Quad platform. Discarded measurements reduce the number of samples and therefore the degrees of freedom in the Student's t-test used to compute the confidence interval. Otherwise, this platform is used the same as the gem5 simulator platform.

A.1 Reproduced Experiments

We generated configurations using the benchmark's RBAConfiguration Java class. For all the configurations, we set the parameters for the role, user, and permission connectivity to sequential (which the benchmark author's call uniform, and have as a parameter value of 1). We set the role hierarchy to 0 for all of the results we report in this paper, meaning no role hierarchy was used. We found that the role hierarchy has little effect on the performance of access enforcement for the SDPs we considered, because the SDP data structure does not traverse the role-permission map.

With the number of permissions at 250, we generated one configuration for the number of roles in {500, 700, 2000, 3000, 6000, 8000, 10000}. We set the number of roles per user to the number of roles, and the number of roles per permission to the number of roles divided by the number of permissions. With the number of roles set at 100, we generated one configuration for the number of permissions in {100, 500, 700, 2000, 3000, 6000}.

We generated session profiles for the above configurations using the modified SessionProfile Java class as described in Section 3.2. For each inter-session configuration, we generated thirty-nine session profiles: three session profiles with 5, 100, and 1000 access requests per session for each of the number of sessions between 2 and 15, All of these session profiles set the number of sessions per access check equal to the number of sessions, activated 3 roles per session, and only issued access requests to resources the user has permission to access with the activated roles. We generated three session profiles—with 5, 100, and 1000 access requests per session made to permitted resources—for each intra-session configuration, and with 15 sessions in each profile, 15 sessions per access check, and the number of roles per session equal to the total number of roles in the configuration. We attempted to match the configuration and session profile parameters to those reported in the prior work, but the number of access requests per session was not available, so it is introduced as a variable in our study.

We ran the benchmark using the above configurations and session profiles with the access matrix and CPOL SDPs on the Intel Core2 Quad. Figure 6 shows the mean access time using the methodology described in Section 3.5. The first row shows the inter-session experiments, with the second and third rows showing the intra-session experiments with varying roles and permissions, respectively. The first column is with 5 access requests per session, and the second and third are with 100 and 1000. All error bars show the 95% confidence interval.

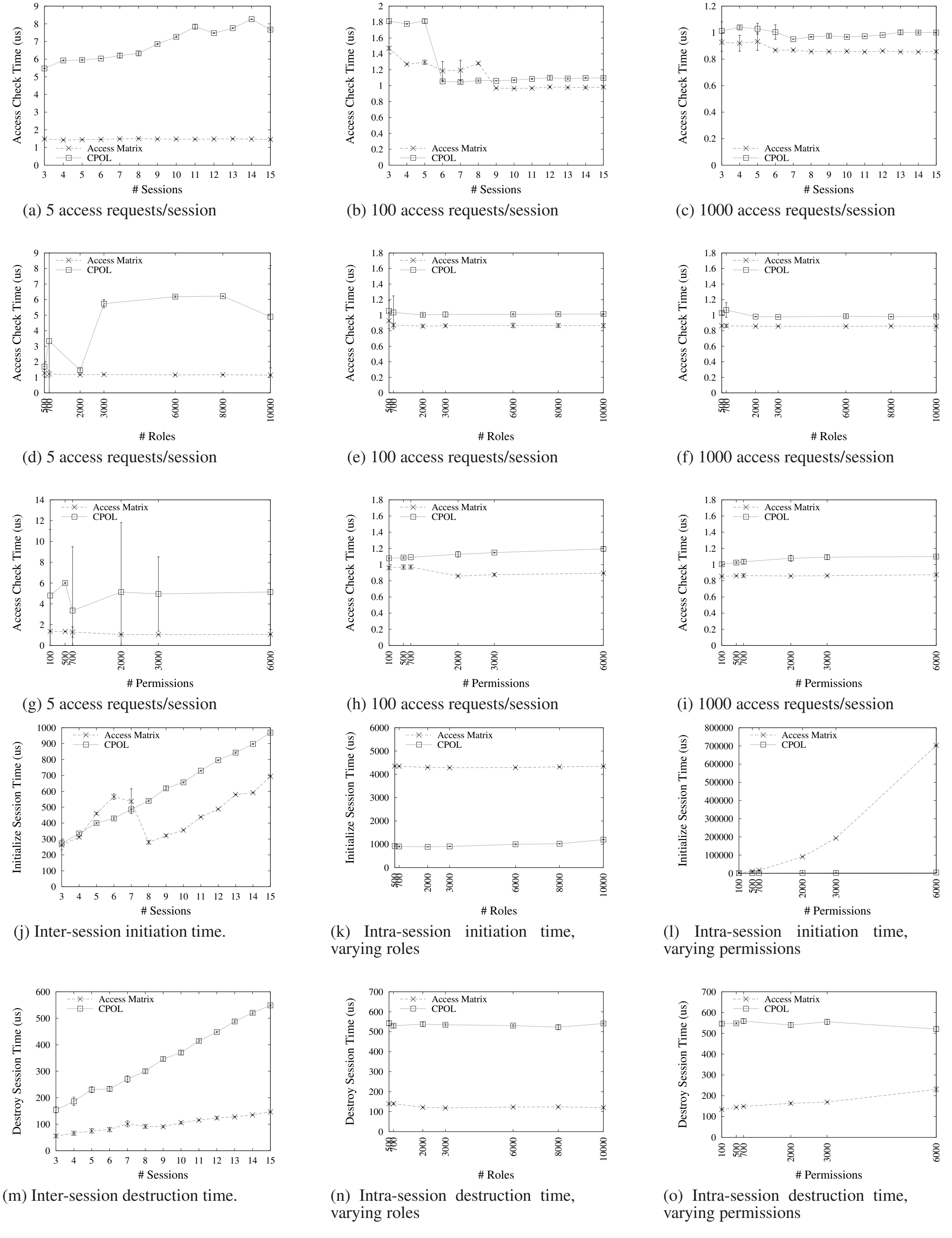
Both SDPs perform best and with least variance when 1000 access requests are issued per session, which was the largest we tried. The long duration of the session enables the benchmark to reach the best steady-state performance sooner with respect to benchmark iterations. With only 5 access requests per session, the access matrix has fairly stable performance around 1.5μs per access request across the inter- and intra-session experiments. CPOL reaches an underperforming steady-state, and the benchmark terminates with a lower performance than that seen with longer sessions. The large confidence intervals for CPOL in the intra-session experiment with varying permissions is due to this failure to reach the steady-state.

In comparison to the prior work, our results show a smaller gap between the access matrix and CPOL access request times when the number of access requests per session is large. The performance of the access matrix is slightly higher than the previously reported numbers, possibly due to differences in the platform or modifications to the benchmark and methodology. CPOL however performed better than the previous work, which we attribute to sensitivity to the number of access requests per session.

A.2 Session Initiation and Destruction

Figures 6j, 6k, and 6l show the total initiation time averaged across all steady-state iterations and VM invocations for each of the experiments. Although these times will not be affected by the session duration, shorter sessions will suffer more due to larger initiation and destruction costs. For these particular measurements, we chose to use 1000 access requests per session. As more sessions are added, we expect a linear increase in the total session initiation time, as seen in Figure 6j. For the inter-session experiments, the average initiation time for a single session with CPOL is between about 60 and 100 μs, and the access matrix is between about 45 and 100 μs. As the number of roles increases, the session costs remain flat because these two SDPs do not operate on roles. Figure 6l shows the inititation time with increasing permissions, for which the access matrix SDP scales poorly. With 6000 permissions, the inititation time per session is over 45 milliseconds.

Figures 6m, 6n, and 6o shows the mean of the total destruction time for the inter- and intra-session experiments. The access matrix SDP seems to have a good destruction time, although the system-wide cost to destroy a session is not captured since garbage collection is not included in these times. For the inter-session experiments, the average time to destroy a single session with CPOL ranges between about 35 and 50 μs, and with the access matrix about 10 to 15 μs. The intra-session experiments have flat destruction times, indicating that the time needed to destroy a session with these SDPs is independent of the number of roles or permissions a session uses.



(a) 5 access requests/session

(b) 100 access requests/session

(c) 1000 access requests/session

(d) 5 access requests/session

(e) 100 access requests/session

(f) 1000 access requests/session

(g) 5 access requests/session

(h) 100 access requests/session

(i) 1000 access requests/session

(j) Inter-session initiation time.

(k) Intra-session initiation time, varying roles

(l) Intra-session initiation time, varying permissions

(m) Inter-session destruction time.

(n) Intra-session destruction time, varying roles

(o) Intra-session destruction time, varying permissions

Figure 6: Experimental results with the modified benchmark