Exploiting Data Parallelism in SELinux Using a Multicore Processor

Arun Kalyanasundaram∗‡, Bodhisatta Barman Roy†‡, Shrisha Rao∗§
∗International Institute of Information Technology Bangalore, India
†National University of Singapore, Singapore
‡arun.k@iiitb.net, §shraro@ieee.org
§bodhi@comp.nus.edu.sg

Abstract—Security Enhanced Linux, popularly known as SELinux [1], is a Linux operating system feature that provides fine-grained access control over system resources. Our goal is to optimize the performance of SELinux for a multicore processor and analyze the efficacy of our approach. We study the architecture of SELinux to identify specific components that cause performance bottlenecks, and empirically validate our claims. We propose various techniques to delegate processor-intensive computations to multiple cores. We perform several experiments to evaluate the performance of our approach under different conditions, and discuss the intricacies of our implementation. Our results show that software applications with security validations have an inherent data parallelism which can be exploited for concurrent execution, but the gain in efficiency depends on design of the application and the hardware platform. In addition, we find that the gain in efficiency is also influenced by the optimization technique and the system configuration. We use a Cell Broadband Engine (CBE) processor running SELinux for our experiments, but our approach can be easily adapted to applications with a similar security validation framework running on other multicore platforms.

Keywords: parallel algorithms, security, operating systems, Security Enhanced Linux (SELinux), cell broadband engine

I. INTRODUCTION

SELinux is a Linux operating system feature that implements the Mandatory Access Control (MAC) security paradigm. MAC operates on a set of rules to constrain a ‘process’ from performing a particular operation on a resource (e.g. file, directory, etc). These rules form the security policy of a system and is centrally controlled by an administrator. Each resource and process is assigned a label called the security context (SC). Thus a rule simply states if one SC is allowed to perform an operation on another SC.

One of the major drawbacks of SELinux is the performance overhead associated with it. The system performance reduces by about 7% when SELinux is enabled and in certain cases like networked systems, the overhead may be much higher [2]. In addition, the worst case performance could be unacceptably low depending on the number of rules in the security policy [3]. Our goal is to optimize the performance of SELinux for a multicore processor and analyze the efficacy of our approach.

In this paper, we first study the architecture of SELinux to determine the scope for introducing data parallelism. We pin-point the specific components within SELinux that cause performance bottlenecks and empirically validate our claims. Based on our analysis, we propose various techniques to modify the SELinux architecture for parallel execution. Our work takes a high-level approach so that our proposed techniques can also be applied to other large-scale security software. More specifically, we optimize the general process of security validations, which involves verifying the credibility of a given entity against a set of trusted entities. We perform several experiments to evaluate the performance of our approach under different conditions. For our experiments, we use the Cell Broadband Engine (CBE) processor as it is known to outperform single-core processors by almost an order of magnitude [4].

The rest of the paper is organized as follows. In Section II, we discuss the background and state of art related to our work. In Section III, we study the architecture of SELinux to identify performance bottlenecks and propose various techniques to achieve data parallelism. In Section IV, we discuss our experimental setup, intricacies/challenges of our implementation, results and inferences. We conclude with a view for future work, in Section V.

II. BACKGROUND AND RELATED WORK

Security has been an integral part of every software application and a primary focus of networking and operating systems research. With the advent of cloud computing, security has gained prominence like never before, since these applications can now be accessible from anywhere in the world and are thus even more vulnerable than their stand-alone counterparts. However, security comes with a steep price in the form of loss in efficiency. Even so, when it comes to building a security feature, efficiency is often given less importance than robustness, a trend that has led to significant performance losses in modern-day systems, where even an anti-virus application can slow down the system by more than fifty percent. This applies even to operating systems’ inbuilt security features like SELinux [1], RSBAC [5], TOMOYO Linux1, and AppArmor2.

With processor clock speeds reaching saturation, extensive research is underway on optimizing existing software applications for multicore processors. Our work aims to exploit data parallelism in SELinux to optimize its performance for multicore processors. There has not been any sizeable effort

1http://tomoyo.sourceforge.jp
2http://wiki.apparmor.net
in optimizing the performance of SELinux [2]. Though there has been some work on optimizing SELinux for 32 or more processors, no further details exist [2]. There have been attempts to extend the SELinux functionality to a distributed setting [6]. Brindle et al. [6] present a distributed policy-management architecture and address several challenges like synchronization of policy updates, managing multi-system policy, etc. In our work, we focus on optimizing SELinux for a single system with multiple processors (or cores), rather than an inter-connected distributed system, which could be a scope for future work. The multicore optimization is also based on implementing parallelized data-structures [7].

SELinux is based on the Flux Advanced Security Kernel (Flask) architecture [8]. It consists of two major components, the Object Manager which enforces security policy decisions and the Security Server which provides security decisions. The Object Manager is responsible for labeling the resources and respond to changes in the security policy [8]. Every access permission request has to be authorized by the Object Manager, and these requests are managed by the Linux Security Module (LSM) framework [9]. LSM consists of a set of hooks at specific points in the kernel where access to a resource is from a system call. These hooks are a set of function pointers which by default do nothing, unless overridden by a security module like SELinux. The return values of these hooks determine whether an access is granted or denied.

When the Object Manager receives a request from the LSM, it extracts the security context of the source and target, to invoke the Security Server with these parameters. The Security Server makes the decision to grant/deny the requested operation based on the rules specified in the security policy. The Object Manager caches recently computed access decisions in a data structure called the Access Vector Cache (AVC). The AVC is a simple hash-table implementation using open addressing. The Object Manager can bypass the Security Server when a requested access decision is available in the AVC. Since the decision-making logic of the Security Server is extremely processor-intensive, use of the AVC can considerably improve performance.

We evaluate the performance of SELinux using a CBE multicore processor, which consists of one Power Processing Element (PPE) and eight Synergistic Processing Elements (SPE) [4]. The SPEs are designed specifically to perform processor-intensive computations. A SPE by itself cannot initiate a computation, only a PPE can initialize an SPE by loading the desired executable binary into the SPEs’ local cache. In addition, data required by an SPE during execution requires the use of Direct Memory Access (DMA) controllers. At the time of SPE initialization, the PPE sends an address of a location in main memory which the SPE can use as a base address to read/write data. In order to access contents of the main memory, the SPE makes a DMA request with the base address, the number of bytes, and the type of access (read or write). Each SPE has one or more dedicated DMA engines that perform data transfers to and from main memory to SPEs’ local caches, concurrent with program execution. In our implementation we also use the CBE SDK which provides flexible APIs which can perform both asynchronous and synchronous DMA requests.

Performing large-scale experiments using SELinux requires the creation of many security policies with different configurations, but creation of policies is a hard and tedious process. In general, it is a job of system administrators to write these policies, which typically contain thousands of rules. Another issue with manual maintenance of large security policies is the detection of loopholes which could lead to possible security breaches. Several tools have been developed to automate the process of SELinux policy configuration [10]. While these tools mainly provide effective analysis of security policies [10], policy creation is still a manual process. Therefore, we created a set of Unix shell-scripts to automatically generate SELinux security policies with a specific access permission patterns.3

III. DESIGN

In SELinux access control, a process can perform an operation (e.g., read, write) on a resource (e.g., files, directories, ports) only when the corresponding rule is specified in the policy. A rule simply states if a security context can perform a particular operation on another security context. Each process and resource on the system are associated with a security context, and the decision to allow or disallow an operation is based on the following:

- The validity of the security contexts of source (process) and target (resource).
- The presence of a rule in the policy that allows the requested operation.

A security context is considered valid if all of its three components (user, role and type) are valid. The validation is performed by extracting each component and matching it against a list of valid contexts. We will show in Section IV-B that the validation step is the major cause for performance overhead in SELinux. We therefore model this step for parallel execution using the SIMD (Single Instruction Multiple Dataset) programming paradigm. Specifically, we look at exploiting data parallelism by dedicating one or more processing units to the matching operation of each component of the security context. We use the principle of SIMD because each processing unit operates on a different data but executes the same set of instructions. Since the security context has three components, we can use a minimum of three processing units considering each component requires one dedicated unit. We propose the following techniques based on the number of processing units and its loading strategy, and compare their relative performances (runtimes).

- Three Processing Units (3U): In this technique we use one dedicated processing unit for each component of the security context, hence a total of three units. In the CBE processor, these processing units are the Synergistic Processing Elements (SPEs), which operate in slave mode (Section II). Each SPE performs a linear search on the list of valid contexts corresponding to each component of

the security context, as shown in Figure 1. The dashed line in Figure 1 shows the direction of traversal.

- **Three Processing Units with Busy Wait (3U/BW):** In general, SPEs are initialized with a set of instructions and a base address of the required data in main memory. To access this data, the SPE has to make DMA requests and wait until the data is received (Section II). However, when the data is not stored in consecutive chunks of memory (e.g., a linked list), the SPE is initialized every time with the address of the next chunk of data, which is the approach we use in 3U (mentioned above). But we observe that the frequent initialization of SPE has a significant performance overhead. Therefore we devise a busy wait technique, in which the SPE is initialized with the address of a unique location in main memory. This location is updated with the base address of the next chunk of data, every time, after the SPE reads it. The details of the implementation are given in Section IV-A.

- **Six Processing Units (6U):** This technique is similar to 3U, except that two dedicated SPEs per component of the security context are used, instead of just one. Thus a total of six processing units are used in this technique. Since two SPEs operate on the list simultaneously, each SPE can process either the odd- or the even-numbered elements of the list, as shown in Figure 2.

- **Six Processing Units with Busy Wait (6U/BW):** In this we use six processing units using the same design as given in 6U, with each SPE operating in the busy wait mode as described in 3U/BW.

IV. EXPERIMENTS

The goal of our experiments was to evaluate the performances of the different techniques proposed in Section III on a CBE processor. For this purpose we used the Sony Play Station 3 (PS3) with Yellow Dog Linux 6.1 (for the PowerPC architecture) installed. The PS3 has eight SPEs, of which six are accessible to a programmer. SELinux is a part of the security module of the mainline Linux kernel since version 2.6, but the default installation of YDL does not contain SELinux. Therefore in our setup, we recompiled the Linux kernel (version 2.6.27) for the PowerPC architecture with the necessary configurations (file-system extended attribute support, security hooks, etc.) to enable SELinux [3].

We used our experimental setup to study the impact of the following two configurable parameters: (1) the number of valid security contexts; and (2) the maximum size of the AVC. The number of valid security contexts is determined by the number of rules in the policy configuration file. Although the AVC significantly improves the performance of SELinux, it also prevents us from accurately measuring the overhead, due to the decision-making logic in the Security Server (see Section II). Therefore, we run our experiments with two different maximum sizes of the AVC: (a) 512 entries (optimal); and (b) 1 entry (minimal). Since the AVC is a hash-table with a fixed number of buckets, a larger size could imply longer chain lengths and hence poorer performance. It has been found through extensive optimization benchmarking that a maximum size of the AVC with 512 entries is optimal to maintain smaller chain lengths [9]. Ideally, in our experiments the minimal size of the AVC should be zero, however the normal functioning of SELinux requires the size of the AVC to be at least one entry.

Broadly, our experiments can be divided into two phases. In the first phase, we evaluate the performance of SELinux using only the single core (PPE) of the CBE processor. This involves measuring the performance with different numbers of rules in the policy configuration file. We use this to establish a performance benchmark and to show that the security context validation step is indeed highly processor-intensive. In the second phase, we compare the performance of the SIMD techniques proposed in Section III with the single-core performance. In each phase, the experiments are performed with both optimal and minimal values of maximum AVC size.

A. Implementation Details

In SELinux, the process of associating a security context to a resource on the file system is called labeling. When the system boots with SELinux enabled, the labeling of the file system takes place based on the settings in file contexts [9]. Once labeled, the security context cannot be changed without authorization. However, malicious programs can sometimes take advantage of weak policy configurations to modify the security contexts of certain files. This could be a significant security breach and may allow undesirable processes to access protected files. Therefore, SELinux provides a functionality to re-label the file-system, which allows us to restore the original security context on system resources without having to reboot the system. The re-labeling process validates the security context of the resource being labeled against a list of valid contexts. This is the same operation that is performed by the security server of SELinux to make access decisions. Hence, in our experiments we evaluate the performance of this re-labeling process to capture the overhead due to security
validations in SELinux. The SELinux user-space libraries (*setfiles*, *libselinux* and *libselpol*) are modified appropriately to incorporate the proposed SIMD techniques. In addition, we devise a set of automated scripts to generate custom file contexts and policies of a desired size. Our complete source code, scripts and installation guide are available online

A major portion of the computation involved in the validation of a security context is string comparison operations. These operations are performed to match the context being labeled with the set of valid contexts. We offload these operations to the SPEs in the CBE processor. However, one of the downsides of using the SPE is that data (the security context string in this case) required for the operation has to be transferred using the DMA controller. Typically the SPE places a DMA request by providing the base address of the data in main memory and the number of bytes. Since a string is a null-terminated character array of variable length, the SPE has to place DMA requests one character at a time until it encounters the null character. This is clearly an inefficient strategy, since the SPE keeps waiting till the entire string has been transferred to its local cache. Instead, each character that is transferred can be compared with the corresponding character of the target string at the same time as the next character is being read. This is based on a well-known technique called double buffering. Figure 3 shows the control flow of this operation performed by an SPE, where *mfc_get* is an API provided by the CBE SDK to make DMA requests. The SPE is initialized with the base address of the input string which is the security context we want to validate. Once the input string is transferred, the SPE then uses double buffering to transfer *str*, as shown in Figure 3. Here *str* represents a particular valid string from the list of valid contexts.

**B. Results**

Figure 4 shows the running times of security context validation, measured during the re-labeling process with increasing numbers of rules in the security policy configuration. We analyze the difference in the rate of increase of running times for the two curves. The curve with minimal AVC shows a considerable increase (by about 40%) in run-time with number of rules in the range [0 - 2000], whereas in the same range, the curve with optimal AVC shows only a negligible increase (by about 8%) in run-time. However when the number of rules is in the range [2500, 4000], the increase in running time is considerably higher in both cases, which is 64% and 112%, for optimal and minimal AVC sizes respectively. An intuitive explanation for this phenomenon is that the hit ratio of the AVC is generally quite high with fewer number of policy rules. This immense drop in performance with minimal AVC size also establishes the fact that security context validations are computationally intensive. In many modern day systems, the number of rules in a security policy is well in excess of 5000, hence even with an optimal AVC size, the performance overhead of SELinux is significant.

Figure 5 shows a histogram of the running times of different techniques with a fixed number of 4000 rules in the security policy. The results show that when an optimal AVC size is used, the running time of the proposed SIMD techniques is higher than the single core run-time. This is counter to our intuition that using more cores should generally improve performance. Our investigation reveals the cause to be an overhead in the SPE, due to: (1) initialization, which is the act of loading an executable binary to the SPEs’ local cache; and (2) data transferred from main memory during execution. We also observe that techniques that use a busy-wait strategy perform reasonably better, since the overhead due to repeated initialization is considerably reduced. For instance, the running times of 6U and 6UBW are 38% and 7% higher than the respective single-core running times, which goes to show the dominant impact of SPE initialization on the overall performance.

The results with the minimal AVC size show a stark reversal from those observed earlier. The running time of techniques that use multiple cores is lower than single core run-time. Though this is as expected, the variation in the efficiency requires further insights. We observe that the sans-busy-wait strategies perform only slightly better than single core, with
Fig. 5. Comparison of running time of different techniques for SELinux security context validation (with 4000 rules in security policy)

an efficiency gain of about 8% and 11% for 3U and 6U respectively. On the other hand, the techniques that use the busy wait strategy considerably outperform both single-core and their SIMD counterparts, with a gain in efficiency of 32% and 43% for 3UBW and 6UBW respectively. Once again the BW strategies show a significant reduction in run-time since it virtually eliminates the more prevalent overhead of SPE initialization. Table I shows the relative gain/loss in efficiency of each of the proposed techniques when compared to the single core performance.

### TABLE I

<table>
<thead>
<tr>
<th></th>
<th>3U</th>
<th>6U</th>
<th>3UBW</th>
<th>6UBW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal AVC size</td>
<td>-26%</td>
<td>-38%</td>
<td>-15%</td>
<td>-7%</td>
</tr>
<tr>
<td>Minimal AVC size</td>
<td>8%</td>
<td>11%</td>
<td>32%</td>
<td>43%</td>
</tr>
</tbody>
</table>

The key take-away from our results is that security validations in general can be modified for parallel executions, but the gain in efficiency depends on the architecture of the software and the hardware platform. Our approach can be easily adapted to applications with a similar security validation framework running on other multicore platforms.

V. CONCLUSION AND FUTURE WORK

Although there has been considerable research in the field of software security, little has been done to improve performance and mitigate related issues that may arise. To deal with the demand of growing security requirements, modern day operating systems provide sophisticated in-built security features, as in Security Enhanced Linux (SELinux). These often result in unacceptably low system performance. We hypothesized that security validations in general can be optimized by exploiting their inherent data parallelisms. We ran several experiments to evaluate the performance of SELinux when modified for concurrent execution. Our experimental setup consisted of automatically generating security policies and measuring the running time involved in SELinux security context validation, on a CBE processor. We obtained contrasting results wherein both gain and loss in efficiency were observed, depending on the technique and the system configuration. Our investigations revealed that performing computations on the SPE came with a significant overhead in terms of loading time and data transfers. However, we found that the overhead in SPEs can be overcome by utilizing them more often and for longer duration.

In general, software applications designed for a uniprocessor system cannot be easily optimized for parallel computing, and even when so modified the gain in efficiency may not be significant. The problem is especially prominent in security-related applications, since the priority is robustness rather than efficiency. We believe our work will inspire future research in building applications that harness the potential of multicore processors.

One obvious extension of our work is to apply the proposed techniques to other security features / applications like TOMOYO Linux, SMACK³, etc., and compare their performances. Evaluating performance on different multicore architectures like GPGPUs, modern PCs and mobile devices could give greater insights into the effectiveness of the approach; e.g., a performance analysis of SEAndroid⁶, a MAC mechanism for the Android platform. Another interesting direction could be to analyze the proposed techniques in distributed platforms like Beowulf clusters and grid networks.

### REFERENCES


³http://schaufer.ca/
⁶http://selinuxproject.org/page/SEAndroid