

Washington University in St. Louis

Washington University Open Scholarship

All Computer Science and Engineering
Research

Computer Science and Engineering

Report Number: WUCSE-2005-55

2005-11-22

Group Scheduling in SELinux to Mitigate CPU-Focused Denial of Service Attacks

Armando Migliaccio, Terry Tidwell, Christopher Gill, Tejasvi Aswathanarayana, and Douglas Niehaus

Popular security techniques such as public-private key encryption, firewalls, and role-based access control offer significant protection of system data, but offer only limited protection of the computations using that data from significant interference due to accident or adversarial attack. However, in an increasing number of modern systems, ensuring the reliable execution of system activities is every bit as important as ensuring data security. This paper makes three contributions to the state of the art in protection of the execution of system activities from accidental or adversarial interference. First, we consider the motivating problem of CPU-focused denial of service attacks,... [Read complete abstract on page 2.](#)

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research



Part of the [Computer Engineering Commons](#), and the [Computer Sciences Commons](#)

Recommended Citation

Migliaccio, Armando; Tidwell, Terry; Gill, Christopher; Aswathanarayana, Tejasvi; and Niehaus, Douglas, "Group Scheduling in SELinux to Mitigate CPU-Focused Denial of Service Attacks" Report Number: WUCSE-2005-55 (2005). *All Computer Science and Engineering Research*. https://openscholarship.wustl.edu/cse_research/971

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

Group Scheduling in SELinux to Mitigate CPU-Focused Denial of Service Attacks

Armando Migliaccio, Terry Tidwell, Christopher Gill, Tejasvi Aswathanarayana, and Douglas Niehaus

Complete Abstract:

Popular security techniques such as public-private key encryption, firewalls, and role-based access control offer significant protection of system data, but offer only limited protection of the computations using that data from significant interference due to accident or adversarial attack. However, in an increasing number of modern systems, ensuring the reliable execution of system activities is every bit as important as ensuring data security. This paper makes three contributions to the state of the art in protection of the execution of system activities from accidental or adversarial interference. First, we consider the motivating problem of CPU-focused denial of service attacks, and explain how limitations of current approaches to these kinds of attacks make it difficult to offer sufficiently rigorous and fine-grained assurances of protection for the execution of system computations. Second, we describe a novel solution approach in which we have integrated fine-grained scheduling decision functions with system call hooks from the Security Enhanced Linux (SELinux) framework within the Linux 2.6 kernel. Third, we present empirical evaluations of the efficacy of our approach in controlling the CPU utilization of competing greedy computations that are either completely CPU bound, or that interleave I/O and CPU access, across a range of relative allocations of the CPU.

Group Scheduling in SELinux to Mitigate CPU-Focused Denial of Service Attacks

Armando Migliaccio*, Terry Tidwell, and Christopher Gill

{armiglia,ttidwell,cdgill}@cse.wustl.edu

Department of Computer Science and Engineering

Washington University, St. Louis, MO, USA

Tejasvi Aswathanarayana and Douglas Niehaus†

{tejasvi,niehaus}@ittc.ku.edu

Department of Electrical Engineering and Computer Science

University of Kansas, Lawrence, KS, USA

Abstract

Popular security techniques such as public-private key encryption, firewalls, and role-based access control offer significant protection of system data, but offer only limited protection of the computations using that data from significant interference due to accident or adversarial attack. However, in an increasing number of modern systems, ensuring the reliable execution of system activities is every bit as important as ensuring data security. This paper makes three contributions to the state of the art in protection of the execution of system activities from accidental or adversarial interference. First, we consider the motivating problem of CPU-focused denial of service attacks, and explain how limitations of current approaches to these kinds of attacks make it difficult to offer sufficiently rigorous and fine-grained assurances of protection for the execution of system computations. Second, we describe a novel solution approach in which we have integrated fine-grained scheduling decision functions with system call hooks from the Security Enhanced Linux (SELinux) framework within the Linux 2.6 kernel. Third, we present empirical evaluations of the efficacy of our approach in controlling the CPU utilization of competing greedy computations that are either completely CPU bound, or that interleave I/O and CPU access, across a range of relative allocations of the CPU.

Keywords. Execution Security, Denial of Service, Group Scheduling, SELinux.

1 Introduction

For a large number of modern networked systems, ranging from personal computing applications to large scale supervisory control and data access (SCADA) systems for power grid control, ensuring the reliable execution of system activities is every bit as important an aspect of system security as ensuring data integrity. The reason for this is that the primary function of these systems is to *use* data to accomplish a purpose and even perfectly secure data integrity is irrelevant unless the data can be used successfully to achieve the intended purpose. Systems with these kinds of execution requirements are increasingly widespread and subject to attack and abuse. For example, as more and more time-sensitive transactions are being conducted over PDA-like cell phones, and as non-critical business information systems are increasingly linked with critical infrastructure control functions and databases in SCADA systems, the risks of adversarial or accidental interference with the *execution* of essential system functions has steadily increased. These time-and-security-

*This work was conducted during a visit to the Center for Distributed Object Computing in the Department of Computer Science and Engineering at Washington University in St. Louis, USA, from March through October 2005, supported by a research scholarship from the Università degli Studi di Napoli Federico II, Italy.

†This work was supported in part by NSF EHS contract CCR-0311599.

critical systems are therefore vulnerable not only to traditional data integrity security threats but also to threats which interfere with the system’s ability to use resources to support computation execution according to specified constraints.

Limitations of current approaches. Despite the importance of trustworthiness for *both* system data access and execution behavior, traditional security approaches have largely treated *execution control* as a separate issue from data security, and have relegated enforcement of execution requirements either to the application developers and system administrators, or to supporting infrastructure such as real-time operating systems, with little attention to the potential benefits of integrating application security requirements, and the security policies and mechanisms that enforce those requirements, more closely. This separation has resulted in the following limitations of the current state of the art in system security: (1) rigid architectural boundaries across which it is difficult to integrate policies and mechanisms for execution control, and (2) the presence of unanticipated “loopholes” that can weaken or even bypass enforcement of desired policies. For example, popular security techniques such as encryption and role-based access control (RBAC) offer protection for system data, but offer only limited protection for the *execution* of computations using that data from significant delays due to accidental or adversarial denial of service.

Research goals and problem statement. The goals of the research presented in this paper are to increase understanding of the role that control over computations’ *execution* plays in system security overall, and in doing so to address the limitations of the current state of the art in system security described above. We have pursued these goals by creating a novel integrated security model for assured execution of critical system computations, and by implementing and evaluating that model in the context of the widely used Linux operating system. The research problems we have addressed in defining this model and developing a prototype implementation of it in the Linux 2.6 kernel include:

1. How to support explicit specification of execution requirements in a way that is both sufficiently flexible to express the unique sets of constraints imposed by a variety of applications, and also sufficiently rigorous to enforce the particular constraints of each application with high fidelity?
2. How to configure efficient policies and mechanisms to enforce execution requirements, so that interference from mis-behaving computations is identified rapidly, the effects of interference on well-behaved computations are minimized, and mis-behaving computations are subjected to application-configurable consequences such as restriction of their available resources?
3. How to integrate fine-grained execution behavior enforcement with existing operating system interfaces and mechanisms in a standard way, so that the impact of feature changes in subsequent releases of the operating system kernel will be manageable in maintaining our security model implementation over the long term?

Solution approach. In this paper we demonstrate that flexible *group scheduling* (GS) techniques [1, 2] which we originally developed for managing real-time tasks with time-varying resource requirements [3], offer a general capability for fine-grain computation control that we can exploit to help ensure the execution of critical system activities. By integrating these execution control capabilities within the SELinux [4] framework, we can achieve a novel synthesis of *access control* (via the SELinux system call hooks), *admission control* (via on-line assignment of feasible execution budgets), and *execution control* (via on-line enforcement of those execution

budgets). As our empirical results in Section 5 demonstrate, this integrated model allows us to protect the execution of specific computations from the computational demands of other computations. We call this kind of protection *execution security*, in that once a computation is admitted to the system, the execution assurances that were given when it was admitted are rigorously enforced even in the face of other computations aggressively attempting to exceed their advertised resource demands.

Prototype implementation and evaluation. To evaluate the effectiveness and efficiency with which our integrated approach can be realized, and its impact on the execution security of different kinds of computations, we have implemented a prototype of this unified model within the SELinux framework of the Linux 2.6 kernel, augmented with the KURT-Linux [5, 1, 2] real-time operating system modifications. We have used the Data Stream Kernel Interface (DSKI) and Data Stream User Interface (DSUI) packages [6, 7] to provide low-overhead instrumentation hooks at key points throughout this framework, allowing us to collect fine-grain traces of system execution behavior. Using our prototype implementation, we have conducted several experiments using different CPU scheduling policies, to control the behavior of competing greedy¹ computations under varying relative CPU allocations and under differing assumptions about whether a computation interleaves I/O and other blocking system calls among otherwise of compute bound phases of execution. Our experimental results in Section 5 show that with the scheduling policies we have investigated so far, the level of protection that we can give to a computation is sensitive both to the relative allocation of the CPU to the computations in the system, and to each computation’s own behavior in exploiting the opportunities it is given to execute. Overall, however, our results show that our approach can be used effectively and efficiently to ensure computations receive their specified execution opportunities, regardless of the CPU demands of other computations.

Research impact. The research presented in this paper offers fundamental advances in the state of the art for design and implementation of trustworthy systems, in which policies and mechanisms for *execution security* can be coordinated using an integrated security model and implementation. The ability to control not only the conditions under which access is granted, but also the conditions under which resources are used, the relative allocations of resources, and the subsequent enforcement of those resource allocations during the execution of computations, offers significant potential improvements in the execution security of time-and-security-critical systems.

The rest of this paper is structured as follows. Section 2 considers the problem of CPU-focused denial of service attacks, which motivates our work on an integrated execution security model and implementation. Section 3 presents our solution approach, in which group scheduling techniques developed in our previous work are integrated with the SELinux framework to provide execution security assurances. Section 4 describes the design and prototype implementation of our solution approach within the Linux 2.6 kernel. Section 5 presents an empirical evaluation of the efficiency and effectiveness of our prototype implementation in protecting computations from the resource demands of other computations. Section 6 describes related work on security and execution control, and Section 7 offers concluding remarks and summarizes issues that remain open as future work.

¹We will use the term “greedy” in this paper to connote a computation that does not yield resources or otherwise self-schedule except to make blocking system calls such as I/O operations. A greedy computation that is CPU bound will make use of all opportunities it is given to execute by the OS thread scheduler, while a greedy computation that makes blocking system calls will only miss opportunities to execute that were given while it was blocked.

2 Motivation: CPU Focused Denial of Service

Denial of Service (DoS) attacks are particularly difficult to mitigate, in part because of the variety of ways in which they can affect the targeted system. Despite having the potential to inflict similarly devastating results, DoS attacks may differ in (1) the means the attacker uses to engage the target machine, (2) the specific resource(s) targeted, (3) the effect on the targeted resource(s), and (4) the location(s) from which the attack is orchestrated. These differences are captured in Shield’s taxonomy [8] of DoS attacks. Shield points out that the primary organizing factor in categorizing these attacks is the resources each attack seeks to compromise or consume in order to interfere with the execution of computations using them. Our research seeks to mitigate a specific sub-category of DoS attacks represented in this taxonomy consisting of *CPU-focused* attacks such as ‘Stream.c’ [9] and ‘Nukenabber’ [10]. In CPU-focused DoS attacks, the attacker attempts to consume all available processing cycles, lock synchronization resources on the target machine, or otherwise deny execution of essential system computations. For example, an attacker could crash a httpd child process in Apache version 2.0.42 [11] by using a specially crafted HTTP packet to cause a null pointer dereference.

CPU-focused DoS attacks require fundamentally different mitigation techniques than those provided by conventional system protection approaches. To an adversary, the advantages of CPU targeted DoS attacks are that they (1) may bypass front-end heavy traffic processing systems used to defend against Distributed DoS (DDoS) attacks, (2) may require application-specific monitoring and enforcement for detection and mitigation, and (3) may require less bandwidth to achieve a greater overall effect.

Conventional approaches to harden systems against DoS attacks may employ a highly redundant or masked front end that dampens the effects of heavy incoming traffic [12]. However, this is no guarantee that a given, possibly critical, process operating behind this front end is not vulnerable to a CPU-focused DoS attack. Such a single point of failure, for example on a server that hosts a SCADA system status database, could be compromised even if the front end is able to drop or redirect maliciously high traffic loads. Therefore, conventional network-based mitigation strategies may fail to protect systems from CPU-focused DoS attacks.

The ‘Nukenabber’ [10] and Apache [11] attacks, and many other CPU-focused DoS attacks, work at higher layers of the system architecture than network-focused DDoS attacks. These two attacks, for example, rely on exploiting vulnerabilities present in off-the-shelf software that runs in user-space. Effectively detecting and filtering traffic targeting these vulnerabilities would require a combination of firewalls aware of application state and sophisticated application proxies. Furthermore, sustained effort would be required to keep such firewalls and proxies up to date, as software upgrades are a fact of life for many systems, and the introduction of new vulnerabilities, *e.g.*, to stack buffer overflows, remains an ongoing concern despite advances in tools and techniques to find and remove those vulnerabilities [13].

The ability of the attacker use very little network bandwidth in launching a CPU-focused DoS attack is another reason CPU-focused DoS attacks can be difficult to mitigate using conventional approaches. For example, Crosby shows how necessary (and ordinarily safe) services can be compromised by specially crafted malicious inputs [14], allowing attacks that can cripple services but that can be mounted using incredibly small traffic loads. This gives the attacker several advantages over network-focused DoS attacks. First, it reduces the potential number of zombie boxes needed, thereby reducing the effort required to mount an attack. In addition, these attacks can be launched across bandwidth-limited networks, such as wireless or cellular systems. In addition, while some DoS attacks may deny service to all systems operating on a shared network, a CPU-focused attack can disable a single system, while leaving all other systems intact. This could be used by an attacker as a way to conduct “man in the middle” or other similar multi-stage attacks

that rely on selectively impersonating particular disabled systems on a shared network. Finally, restricted use of bandwidth may make an attack harder to detect, as a small number of packets from disparate sources could be used to mount the CPU-focused DOS attack.

All of these factors suggest that defense against CPU-focused DoS attacks is very difficult using *only* conventional network based mitigation strategies and common system configuration options to protect vulnerable systems. For example, use of the *ulimit* setting in Linux has been discussed in security-related on-line mailing lists as a way to mitigate “fork bomb” [15] style attacks. With more selective CPU-focused attacks like the ones noted by Crosby, however, the attack does not rely on actions that are easily detected and then can be limited or prevented, such as spawning an ever increasing number of threads. Instead, a CPU-focused attack can insidiously cause a normally well-behaved thread to consume computing resources inappropriately. Addressing problems such as this will require the ability to detect anomalous computation behaviors just as current approaches detect anomalous network behaviors.

Furthermore, since attacks of the kind identified by Crosby make use of specially crafted malicious inputs, it is unlikely that all potentially problematic inputs to a program can be modeled. Attempts to automate identification of problematic program inputs in general are limited by the halting problem, and special cases that are computable may still be intractable for reasonably sized systems: for example, permutations of fixed sequences of inputs for a computation that is known to finish for all inputs.

In light of these concerns, it is unlikely that reliance on network-level DoS mitigation approaches or existing system configuration options alone will be able to offer adequate execution security in the face of malicious CPU-focused DoS attacks. These limitations of the current state of the art motivate our solution approach presented in Section 3, in which we support defense-enabling scheduling strategies within the operating system kernel, to ensure specified allocations of resources are enforced even during CPU demands by some computations that would otherwise result in denial of service to other computations.

3 Solution Approach

Consider a server system which spawns a thread to execute primarily CPU bound computations in response to each client request. Now, assume that at least one of the client request types exhibits a vulnerability which can cause the server thread to consume excessive CPU cycles when attempting to handle malformed requests. Such a vulnerability makes it possible to create a CPU based DoS attack on this server using very low request volume and network traffic by providing pathological service requests. In this scenario the attacker could use a set of zombie machines to generate the pathological messages, but would not necessarily need to generate a high volume of such requests for the attack to succeed. Our approach helps defend against such attacks by integrating the KURT-Linux group scheduling framework with SELinux to (1) provide fine-grain control of execution behavior, and (2) add policies and mechanisms for detection and handling of execution constraint violations, to the existing access violation control policies and mechanisms provided by SELinux. Under our approach system administrators could thus establish constraints defining acceptable behavior for each service type. Then, if pathological requests were sent to exploit an existing vulnerability, the violation of execution constraints by server threads can be detected and their behavior can be controlled.

3.1 Overcoming Limitations of Previous Approaches

Previous approaches to security and execution control exhibit limited integration. On the security side, SELinux limits itself to essentially a binary test of whether a given action, such as spawning a child thread, is permitted or not. On the scheduling side, no

consideration of the security context, and in particular no consideration of security related behavioral history is made at all. Our approach moves beyond the limits of current methods to create a framework involving direct integration of the SELinux access control and the group scheduling execution control infrastructures.

The KURT-Linux group scheduling framework expands the computation control semantics which can be configured significantly beyond the classical dynamic (generic) and static (real-time) POSIX priority semantics provided by the vanilla Linux kernel. The group scheduling framework does this by (1) supporting the collection of threads and other computation elements into groups, and (2) expressing a variety of desired execution semantics through scheduling decision functions for each group. For this discussion the most relevant use of a group is to describe the set of components that must be executed to complete one or more application computations. Each group can have an arbitrary scheduling decision function associated with it to select among group members according to the semantics of the computation(s) represented by the group. Composition is supported by permitting groups to be group members, and so sets of groups representing computations are composed into a system-level scheduling decision hierarchy which determines which of the computations in the system should use the CPU whenever a scheduling decision is required. The group scheduling model [1, 2] provides application state aware scheduling semantics [3] which users can leverage to implement scheduling decision functions that are capable of controlling fine-grain execution behavior of all computations in the system, using any and all available information.

Within this framework it is thus straightforward to implement scheduling decision functions which can take information provided by the SELinux subsystem into account when controlling system computations. The more difficult aspects of this issue are, however, (1) deciding what information is relevant, (2) implementing mechanisms which collect the desired information, and (3) making that information available to the security-aware scheduling decision functions. However, while this effort is both non-trivial and context dependent, our experience shows that these three requirements can be met in a variety of contexts [3], including the execution security domain we consider in this paper.

3.2 Integration of Usage Policies and Penalties

The operating system plays a fundamental role in our approach to execution security. We have integrated the configurable group scheduling capabilities of the KU Real-Time (KURT) extensions to Linux [16, 1, 2], with the Security Enhanced Linux (SELinux) [4, 17] features available in the Linux 2.6 kernel, as a foundation for role based access, admission and execution control. In our prototype implementation we have focused on several basic strategies for providing fine-grain, selective, and coordinated configuration of (1) access, admission, and execution control decision semantics and the kernel-level mechanisms used to implement them, and (2) instrumentation of kernel level operations relevant to evaluating system security and distribution of detailed audit streams for both on and off line forensic analysis.

SELinux already supports a type enforcement model as its primary method of enforcing security semantics, using role-based access control (RBAC) as a secondary security model and as a method for describing constraints on system operations [4, 17]. The existing SELinux mechanisms do not, however, integrate access, admission and execution control. At the kernel level in particular, the semantics of multiple system components may affect the overall access, admission, and execution control semantics that can be enforced. Processes and threads are the only computation types currently considered under the SELinux control model, but interrupt handlers, soft-IRQs and tasklets also perform computational functions affecting the behavior of application programs. Our approach uses the KURT-Linux Group Scheduling framework [3, 2, 1] to support execution control over *all relevant* computation types.

Exposing kernel mechanisms to control. A major theme of our previous work on KURT-Linux in general, and on group scheduling in particular, has been to achieve a systematic increase in the scope and precision of control over computational activity in the endsystem. We have exposed a growing set of kernel and middleware mechanisms to control, and have designed programming models to make this control as transparent and as configurable as possible. The research described in this paper has focused on integrating existing kernel-level security mechanisms in SELinux with our existing group scheduling framework by extending the SELinux security related decision hooks that exist throughout the Linux kernel to incorporate calls to the group scheduling framework. This means that (1) fine-grain decisions about computation execution can be considered under the system security model and (2) timely execution of security-related decision functions can be considered under the system scheduling model.

Group scheduling model. The essential features of the group scheduling model are: (1) the ability to *group* application components with supporting elements of the system infrastructure; (2) the ability to associate policies, implemented by scheduling decision functions, with these groups for specification of how resources should be used in support of the group members; (3) the ability to compose groups to form a unified system-wide policy for resource management, potentially customizing control of everything from lowest level interrupt service routines to highest level coalitions of inter-operating applications; (4) the ability to enforce the specified resource use semantics as defined in the composed group policy structure; and (5) the ability to monitor system behavior at the same level of detail at which the application specifies its requirements, and to use the measured information as feedback to the enforcement functions.

A given application component can be associated with multiple groups because the policies controlling the system may have more than one possible reason to choose it for execution. Similarly, system infrastructure elements may belong to multiple groups because the services they provide are used in different resource management contexts. For example, a thread pool might be used to service events from a given device, and the device service routine might thus be a member of each threads' group. Alternatively, the device service routine might be a member of a group structure controlling how the device is shared (arbitration) among several computations. Hierarchical organization of groups, possibly crossing resource administration boundaries, allows schedulers to *weave* the policies controlling multiple applications into a composite system-level policy. The granularity at which this weaving is done can be configured flexibly, and at its finest resolution can approach the inherent limits of the system in both semantics and time granularity.

Our existing kernel-level and middleware-level implementations of the group scheduling model for KURT-Linux [3] offers a suitable foundation for the scheduling enforcement needed for execution security assurance. However, we used our kernel level group scheduling implementation for the prototype implementation of our execution security model, because (1) it offers significant performance advantages due to reduced overhead of switching between kernel and user space [2], and (2) it allows direct integration with the SELinux system call hooks, making it more difficult for an attacker to interfere with the system's control over access to system calls, allocation of resources, and execution of computations.

Two results of our previous evaluations of the group scheduling framework [3] are important to the work described in this paper: (1) the ability of group scheduling to accurately isolate resource (CPU) use by a critical stream processing computation from interference from the use of that resource by competing (non-critical) streams; (2) the ability of group scheduling to rigorously and dynamically enforce developer-defined constraints (stream frame progress) on behavior of non-critical streams, in contrast to the default Linux scheduler priority based semantics or to static CPU allocation techniques.

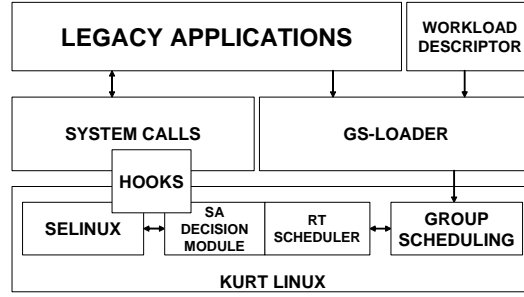


Figure 1. Integration of SELinux and Group Scheduling

Integration of security and scheduling. The integration of security and scheduling mechanisms at the kernel level means that resource access, admission and execution control, while logically distinct decisions, can be made to interact or can even be merged with very high efficiency. For example, if one thread within a computation attempts a forbidden resource access, one possible adaptation that the system designer may configure is to impose scheduling consequences for the offending thread (and possibly other threads associated with it) under the group scheduling framework.

4 Prototype Design and Implementation

The modular architecture of our solution approach is illustrated in Figure 1. KURT-Linux in *enforcement mode* (i.e., the SELinux security server enforces security decisions) is the foundation of our solution architecture. We apply the KURT patches to the Linux operating system in order to:

- *Provide basic mechanisms to control resource allocation:* our first objective is to provide a mechanism to control CPU allocation. To this end, the *Group Scheduling Framework* module implements different scheduling decision functions (SDFs) to (1) allow the user to choose from a variety of scheduling semantics, (2) ensure accuracy in CPU allocation, and (3) ensure predictability in computation execution. The set of SDFs we have implemented for the research described in this paper provides both *open loop* and *closed loop* control of CPU usage. Our open-loop SDFs are based on the weighted round-robin algorithm [18, 19], so that periodically each process is given a certain time quantum. Our closed-loop SDFs provide feedback control over the CPU percentage used by each computation. The scheduler controls the progress of each computation, preventing it from using more than its allocated share of the CPU. Section 4.1 gives details about our prototype implementation, and Section 5 presents an evaluation of the precision and overhead of these strategies.
- *Integrate SELinux and Group Scheduling:* SELinux and group scheduling are integrated via system call *hooks* and via the *Security-Aware Decision Module* (SA-Decision Module). Hooks are implemented to keep track of *security-sensitive* information in more detail. This is required since the SELinux security server allows each program to do whatever the security policy specifies, like opening files and sockets and spawning child processes. However, SELinux does not supply mechanisms to track *how* a program uses the system features it is allowed to access – does it open too many files, send excessive network packets, or spawn too many child processes? Based on this information and on decisions made by the SELinux security server, resource allocation policies may adapt to favor *good* processes over *bad* processes. Our prototype implementation provides a strategy to implement such security-aware scheduling of the CPU, as we describe in Section 4.2.

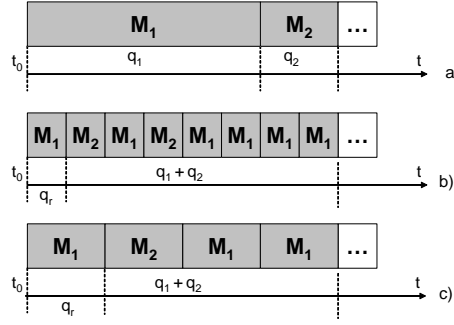


Figure 2. Round-Robin Quantum Allocation Strategies: a) Coarse-Grain; b) Fine-Grain; c) Medium-Grain

- *Allow legacy applications to take advantage of the platform:* the idea behind our approach is to allow legacy applications as well as platform-aware applications to take advantage of its facilities. Each program is associated with a description of resources needed to accomplish its computations. Each computation may describe such requests in terms of the resources needed by the program (such as the CPU percentage, the network bandwidth, or the memory size) and specify a strategy to subdivide a particular resource among its child processes. A *workload descriptor* contains the resource specifications for programs that need to run on the platform. The *Group Scheduling Loader* (GS-Loader) configures the underlying platform (*i.e.*, building up the group SDF hierarchy), and then executes programs listed in the workload descriptor. In our prototype implementation, a resource set may describe the amount of CPU time needed by the program’s computations and the strategy for subdivision of that time among its spawned children, as we describe in more detail in Section 4.3.

4.1 Group Scheduling Decision Functions

A major design goal of the group scheduling framework is to provide maximum flexibility to the system designer in configuring execution control semantics. We have implemented a set of scheduling strategies according to the following approaches:

Open loop scheduling. In this approach, scheduling decisions are made at initialization time, so that the order in which computations are scheduled is prearranged, as is the CPU time allocated to each computation in turn. Our open loop scheduling strategies are based on the weighted round robin algorithm [18, 19]. Each group member can run up to but cannot exceed its quantum within a round of the scheduling algorithm, and IO and synchronization operations, or any other potentially blocking operations, can cause a computation to waste part of the time quantum available to it within a single round. The open loop approach provides a guarantee of execution *opportunity* but not of *utilization* of that opportunity.

Among the possible open-loop quantum allocation strategies, we have focused on those illustrated in Figure 2. In the first strategy, called “coarse-grain round robin,” each group member’s computation may progress up to the expiration of its total allocation, so that the group member is not preempted by the scheduler within that period of time. Afterward, the scheduler picks the next eligible group member from the queue and lets it run for its entire allocation. In the second strategy, called “fine-grain round robin,” the scheduler lets each group members’ computation progress for a prearranged time slice, and then preempts that group member in order to allow another group member’s computation to progress. Once the member’s total allocation has expired, it is no longer allowed to run within the current scheduling round. The third strategy is similar to the second one, except that the time slice is set to the greatest common factor (GCF) of all the group members’ quanta. As Figure 2.c shows, such a choice can minimize the number of switches between

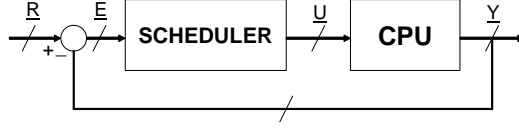


Figure 3. Closed Loop Scheduling Strategy Control Schematic

computations and, at the same time, it can allow the computations to progress fairly. Although the strategy 2.c strikes a compromise between the strategies shown in Figures 2.a and 2.b, in this paper focus only on the first two, since results obtained for them serve to bound the extremes of all possible results. The coarse-grain round robin strategy is suitable for CPU-bound computations since they easily consume the whole quantum assigned to them. IO-bound computations might miss the opportunity to consume their entire allocation due to blocking operations, so they make be better served by the fine-grain round robin strategy thanks to greater distribution of the execution opportunities across the scheduling round. While it is expected that the coarse-grain round robin strategy is more efficient due to fewer forced context switches between group members, the fine-grain round robin strategy is expected to be more forgiving when a group member fails to use part of its allocation.

The relationship between the CPU percentage offered to a group member and the time quantum, is described by Formula 1:

$$\%_i = \frac{Q_i}{\sum_j Q_j} \quad (1)$$

where Q_i is the time quantum assigned to the i^{th} group member, and $\%_i$ is the maximum CPU percentage that it can use. Our experiments described in Section 5 show that Formula 1 is also a reasonably good approximation of the CPU *utilization* by (especially) CPU bound and (to some extent) mixed IO/CPU bound computations. With any of these round robin scheduling strategies, it is possible to protect well behaved computations from misbehaving computations by modifying the time quantum assigned to the misbehaving ones at run-time, to minimize the CPU percentage that they take away from properly behaved computations (those satisfying their execution behavior constraints if any are specified). Our empirical results presented in Section 5 show how the performance and overhead of these strategies varies according to the computations use of the CPU and I/O.

Closed loop scheduling. in this approach, scheduling decisions are made dynamically, and the CPU percentage that each computation is given is adjusted according to specified constraints. This approach can be modeled by the abstract feedback control scheme shown in Figure 3. As the Figure shows, the CPU is the system to be controlled. Inputs are modeled as a vector $\bar{U}_i = (u_1, u_2, \dots, u_n)$ where $u_j = 1$ if $j = i$, and $u_j = 0$ if $j \neq i$. By means of the vector \bar{U}_i , the controller states that the i -th computation has to get the CPU within the next sampling period. Outputs are modeled as the vector $\bar{P} = (p_1, p_2, \dots, p_n)$, where p_i is the CPU percentage consumed by the i -th computation, while the reference is modeled as the vector $\bar{R} = (r_1, r_2, \dots, r_n)$, where r_i is the CPU percentage that a computation must get. The control law can then be expressed as follows:

$$\bar{U}_i : \begin{cases} |\bar{R} - \bar{P}| \geq 0 \\ |\bar{R}_i - \bar{P}_i| = \max_j \{ |\bar{R}_1 - \bar{P}_1|, \dots, |\bar{R}_n - \bar{P}_n| \} \end{cases} \quad (2)$$

where $\bar{R}_i = \bar{R} \cdot \bar{I}_i$ and $\bar{P}_i = \bar{P} \cdot \bar{I}_i$. Formula 2 states that the scheduler chooses as the next computation, the one whose margin between the actual CPU percentage and the reference percentage is the greatest; the choice must be made under the constraint that no computation gets more than its specified reference percentage. This strategy has two fundamental design parameters: the *sampling period*, t_s , i.e., how often the control law is computed; and the *reference period*, T , over which the CPU percentage is computed. The accuracy of the scheduling strategy is affected by the ratio, T/t_s . Figure 4 shows an example where the ratio $T/t_s = 10$; in such a case, each time that the scheduler picks a group member and runs it, the computation may get up to the 10 % of CPU time. The accuracy of the control strategy in this case is thus on the order of 10%. The smaller the ratio T/t_s , greater the overhead, but the greater the precision with which the scheduler can control the computations.

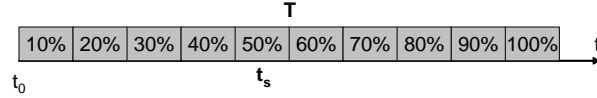


Figure 4. Sampling Period vs. Horizon for CPU Percentage Evaluation

4.2 Security-Aware Decision Module

SELinux provides the LSM framework [17, 20], which is a security *infrastructure* that can be used in conjunction with loadable kernel modules to *implement* desired models of security. The LSM framework adds security fields to kernel data structures and inserts calls to hook functions at critical points in the kernel code to manage the security fields and to perform access control. Each LSM hook is a function pointer in a global *security_ops* table. The *security_ops* table is initialized with a set of hook functions provided by a default security module that provides traditional *superuser* semantics. A *register_security* function is provided to allow a security module to set *security_ops* to refer to its own hook functions. This mechanism is also used to set the primary security module, which is responsible for making the final decision for each hook.

The security server provided with SELinux implements a combination of role-based access control (RBAC), a generalization of type enforcement (TE), and optionally multi-level security (MLS). The access vector cache (AVC) provides interfaces to the hook functions for efficient checking of access permissions, and provides interfaces to the security server for managing the a cache of security permissions (in the form of *access vectors*). The hook functions manage the security information associated with kernel objects and perform access control checks for each kernel operation. The hook functions call the security server and access vector cache to obtain security policy decisions and apply those decisions to label and control kernel objects.

Using System Call Hooks as Interceptors. Figure 5 shows the path of function invocations from the invocation of a system call to the security checking and enforcement functions performed by SELinux. Each system call has a hook which can grant or deny the operation. Instead of providing a new security module from scratch, the Security-Aware Decision Module, along with extra function hooks, integrates the SELinux security module in order to monitor processes' operations and to collect security-sensitive information which may deflect potential situations where a process is misbehaving or malfunctioning. By example, fork bomb attacks [15] could be detected by the excessive number of *fork()* calls, whereas ICMP flood attacks could be detected by the excessive number of opened sockets and/or high rate of packets sent/received. Once these situations are detected, the idea behind the proposed approach is to mitigate their effects by making the scheduling strategy aware of them. The scheduler may then penalize a misbehaving process by

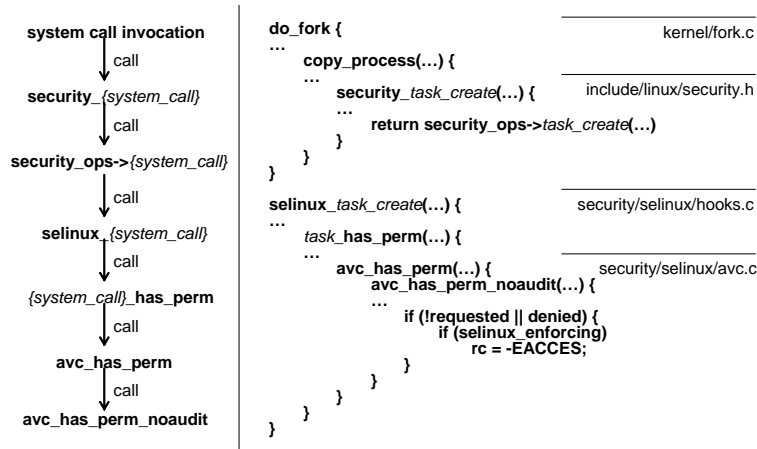


Figure 5. SELinux Call Stack and Function Hooks for *fork()* Example

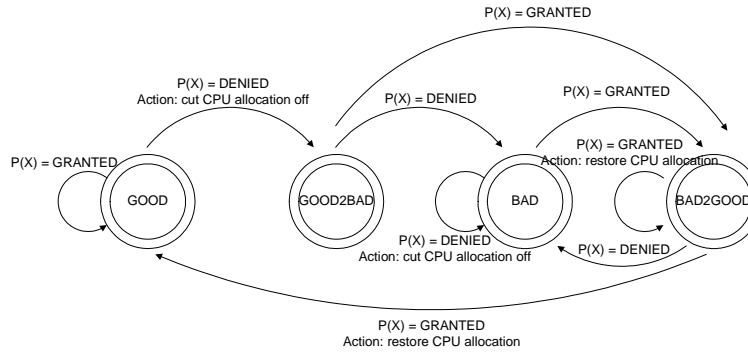


Figure 6. Security Aware Scheduling Strategy

cutting its CPU percentage/time quantum. Depending on the kind of system and its policies, the penalty may be more or less drastic and a process may even be given the possibility to *redeem* itself. Figure 6 shows how a reversible status reduction strategy may be applied in response to misbehavior. X is a set of behavioral attributes tracked via *hooks*, while $P(X)$ is a boolean “permission” predicate evaluated by the SA-Decision Module. The predicate is evaluated every time a security decision is made by SELinux: such a security check can either be *granted* or *denied* if bad behavior is detected. A process can be in the following statuses: (1) **GOOD** - no action was denied and a security attribute check was positive; (2) **GOOD2BAD** - either an action was denied or a security attribute check was negative; (3) **BAD** - either a security attribute check was still negative or an actions was still denied; (4) **BAD2GOOD** - no action was denied and a security attribute check was positive. The resource use reduction strategy is handled by binary division of the CPU allocation during **BAD** status, and recovered through binary multiplication during **BAD2GOOD**.

4.3 Group Scheduling Loader

Any of the scheduling strategies described in Sections 4.1 and 4.2 can be used with the group scheduling framework provided by KURT-Linux. It is desirable that our solution approach can be used to control unmodified legacy programs. In fact, many threats might be hidden in legacy applications, or generally speaking, in applications whose code is unknown and potentially vulnerable to malicious exploit. Running legacy programs under group scheduling control is made possible via the GS-Loader. In our prototype implementation, a legacy program is put under group scheduling control and is then scheduled with respect to its resource set descriptor, which specifies the scheduling strategy (*i.e.*, either closed loop or open loop), the CPU allocation (*i.e.*, either a CPU percentage or a time quantum) and the strategy for subdividing the CPU allocation among the spawned children. We have implemented two subdivision

strategies so far: (1) `FIFO_SPLIT` - this strategy halves the CPU allocation between the parent and child processes, so that any further children spawned from the parent will get their allocation from the parent's already halved allocation; and (2) `SHARE_SPLIT` - this strategy shares the CPU allocation between the parent process and spawned children equally. When a process is terminated, its share is returned to the parent if the parent is still active, or otherwise to the parent's other children, if any.

5 Empirical Evaluation

In this section we present a set of experiments we have conducted to evaluate the precision and overhead of the open loop and closed loop scheduling strategies discussed in Section 4.1. We have also evaluated how the precision and overhead change between CPU-bound processes and processes with a mix of CPU and I/O bound phases, as well as how scheduling precision and overhead change across different relative CPU allocations. To this end, the design of our experiments is as follows:

- *Open-Loop Management*: for each experiment we vary the following parameters:
 - Quantum management strategy: we have tested strategies shown in Figures 2.a and 2.b in Section 4.
 - Type of process: we have measured the execution time of both CPU bound and mixed CPU-IO bound processes. The process' period is set according to the time quantum given to the process. For example if the time quantum is X seconds, the process' period will be less than X seconds to ensure contention.
 - CPU allocation: in each experiment we schedule two competing processes of the same type, but with different relative CPU allocations. We varied the relative CPU allocation between a 5%-95% split and a 50%-50% split.
- *Closed-Loop Management*: for each experiment we vary the following parameters:
 - Type of process: we have measured the execution time between context switch events of both periodic CPU and mixed CPU-IO bound processes. The process' period is set according to the CPU percentage given to the process. For example, assuming that the scheduling algorithm computes the CPU percentage on a reference period of T seconds, the period of the process which gets the $P\%$ will be less than $P \cdot T$ seconds.
 - CPU allocation: in each experiment the scheduler deals with two processes of the same type, but with different relative CPU allocations, which we have varied the CPU allocation from 5%-95% to 50%-50%.

As one indicator of the precision in scheduling control, we have chosen the probability distribution function (pdf) of the process' execution time between context switch events. The accuracy of the scheduling strategy is reflected by the distribution's shape. Tight distributions reflect accurate strategies and vice versa. The percentage of CPU time that each process consumes is another indicator which may reflect the precision of the scheduling strategy at a coarser temporal scale. The margin between the ideal percentage values and the actual utilization gives us a gauge of the precision of the scheduling strategy in allocating CPU use to specific processes.

As overhead indicators we have measured the context switch time, *i.e.*, the time spent by the scheduler to select a new process to run and to schedule it, and the number of context switches that occur per second. In this context, the context switch time is largely determined by the time required to evaluate the group scheduling hierarchy. Both metrics are useful to estimate the basic overhead

added by group scheduling, and how the overhead varies across different allocations of the CPU. The platform used in the experiments was a P4 2.5GHz CPU machine with KURT-Linux modifications applied to Linux kernel release 2.6.11-5.

5.1 Empirical Results

The results presented in this section provide several views of how processes behave under coarse grain open loop, fine grain open loop, and closed loop scheduling strategies. We measured the behavior of two types of processes according to three different metrics to evaluate the effectiveness of our approach: execution times for each dispatch of a scheduled processes, context switch times, and context switch rates.

Figure 7 shows graphs of the distribution of execution times for CPU bound (Figures 7a, 7c, and 7e) and Mixed CPU-I/O processes (Figures 7b, 7d, and 7f). In these graphs, the X axis is a normalized execution time, and thus centered around zero. The Y axis is the set of CPU percentage allocations assigned to the process, which range from 5 to 95 percent. The Z axis in these graphs is the probability density, which is thus in a range from 0.0 to 1.0. The CPU bound graphs give us a view of how precisely the group scheduling framework can control process execution, while the mixed CPU-I/O graphs show how blocking from I/O can introduce variations in process behavior.

Figure 8 shows the distribution of the time required to evaluate the group scheduling decision hierarchy, to choose the next process and schedule it, for CPU bound (Figures 8a, 8c, and 8e) and Mixed CPU-I/O (Figures 8b, 8d, and 8f) processes. The X axis is the evaluation and scheduling time in CPU cycle units on the experimental platform, while the Y axis is the set of relative CPU allocations between two competing processes ranging from (5, 95) to (50, 50) in increments of 5 percent. The Z axis is the probability density. These graphs indicate the overhead we pay for the precise control provided by the group scheduling framework.

Figure 9 shows the distribution of the context switch rates for CPU bound (Figures 9a, 9c, and 9e) and Mixed CPU-I/O (Figures 9b, 9d, and 9f) processes. The X axis in these graphs is the number of context switches per second, while the Y axis is the relative percent CPU allocation split between two competing processes, which ranges from (5, 95) to (50, 50). The Z axis is the probability density. These graphs give another view of the overhead we pay to obtain precise process control.

The next four figures compare results for the open loop (“a”) and closed loop (“b”) scheduling strategies. Figure 10 shows actual CPU use on the Y axis with the reserved CPU percentage on the X axis for both CPU bound and mixed CPU-IO processes. The graph also shows the ideal (diagonal) trend line. The actual utilization by a process is a measure of the scheduling precision. For example in the open-loop scenario, the CPU utilization should observe Formula 1 in Section 4, whereas in the closed-loop scenario, the CPU utilization should follow the reference percentage.

Figure 11 shows the context switch rate on the Y axis and the allocated CPU percentage for the process on the X axis, for the same sets of scheduling methods and processes. A comparison among the context switch rates observed under different scheduling strategies gives us an indication of the trade-off between scheduling overhead and precision of control. Figure 12 plots the difference in actual CPU percentage use on the Y axis and the difference between the intended CPU use for same two competing processes under the open and closed loop scheduling methods. Figure 13 plots the total execution time in microseconds on the Y axis and the percentage of the CPU allocated to the process on the X axis for the same set of processes and scheduling methods.

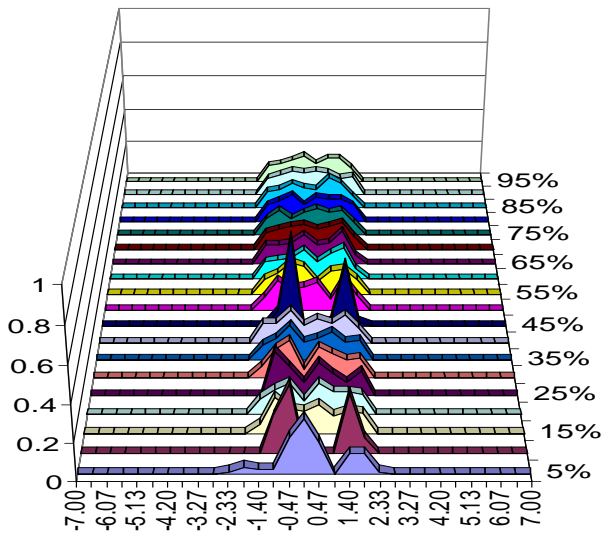


Figure 7a. Coarse Grain Open Loop (CPU Bound)

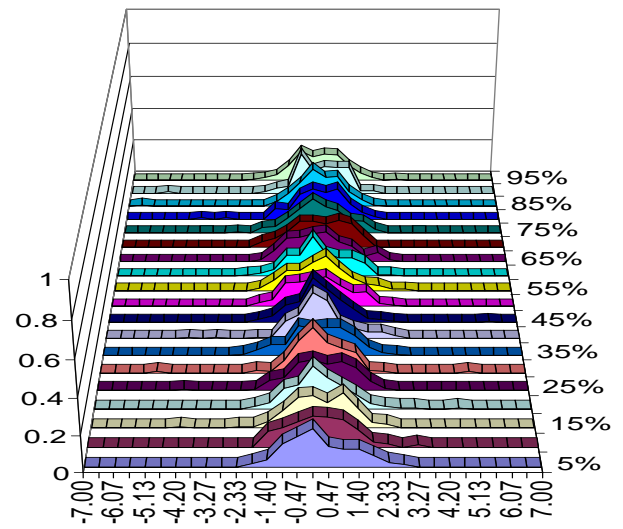


Figure 7b. Coarse Grain Open Loop (Mixed CPU/IO)

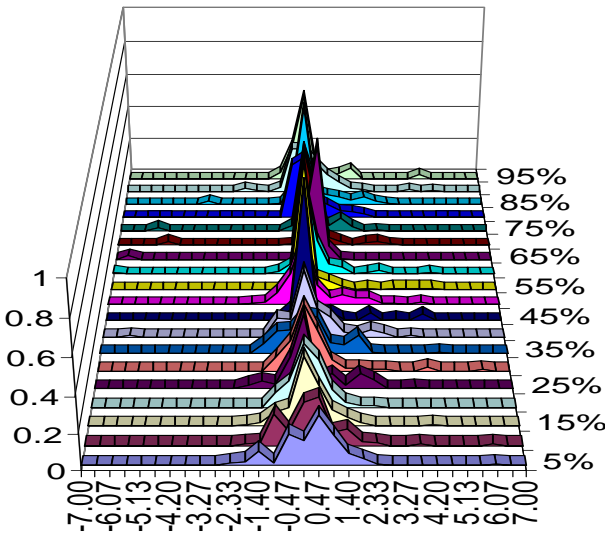


Figure 7c. Fine Grain Open Loop (CPU Bound)

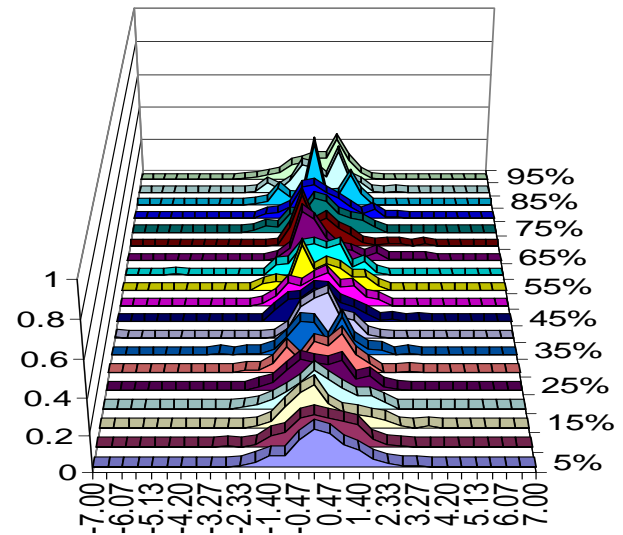


Figure 7d. Fine Grain Open Loop (Mixed CPU/IO)

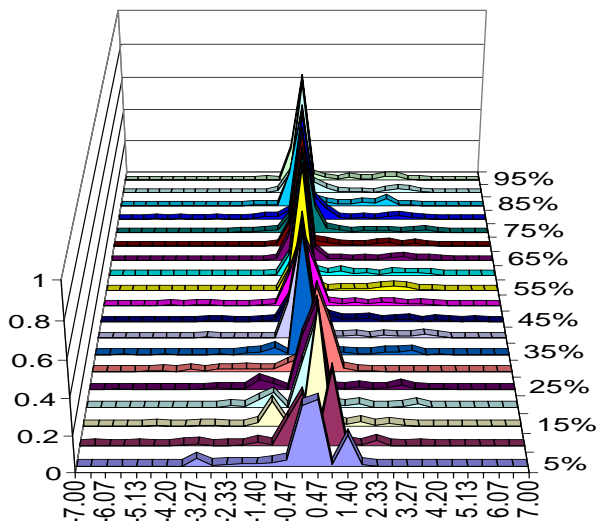


Figure 7e. Closed Loop (CPU Bound)

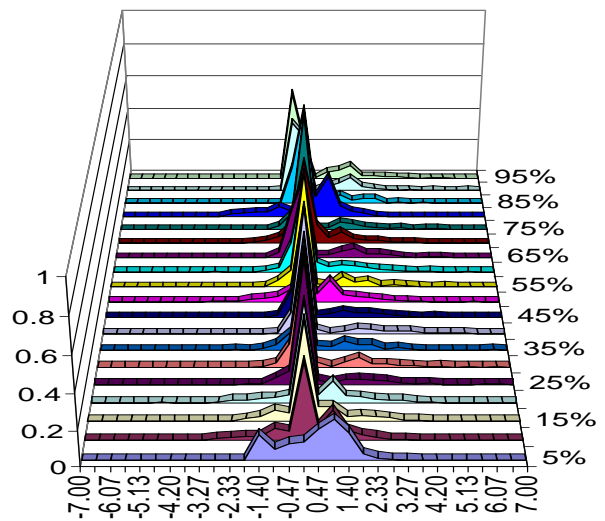


Figure 7f. Closed Loop (Mixed CPU/IO)

Figure 7. Execution Times

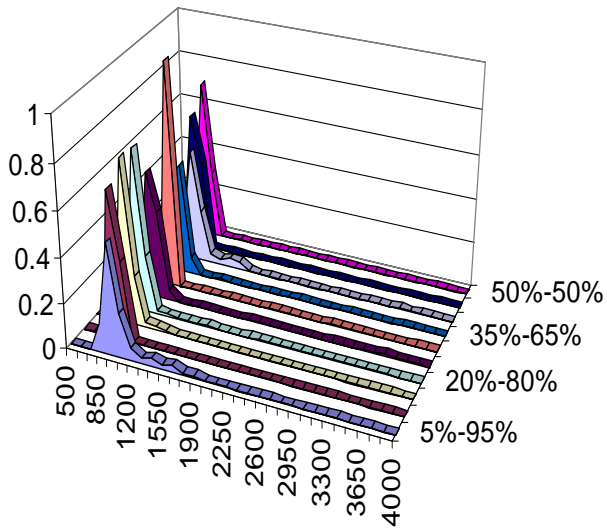


Figure 8a. Coarse Grain Open Loop (CPU Bound)

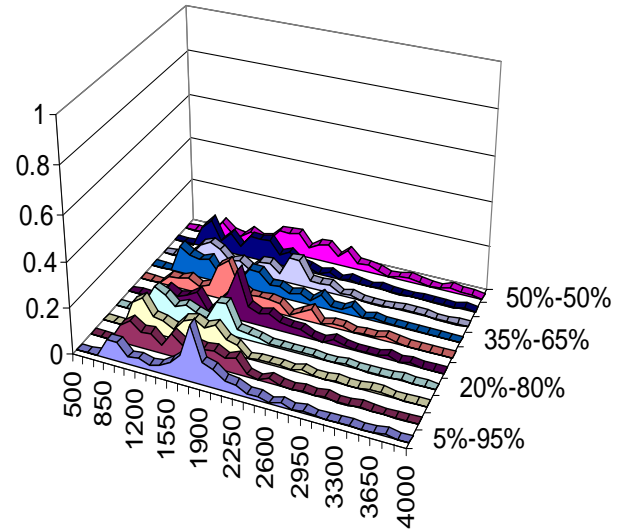


Figure 8b. Coarse Grain Open Loop (Mixed CPU/IO)

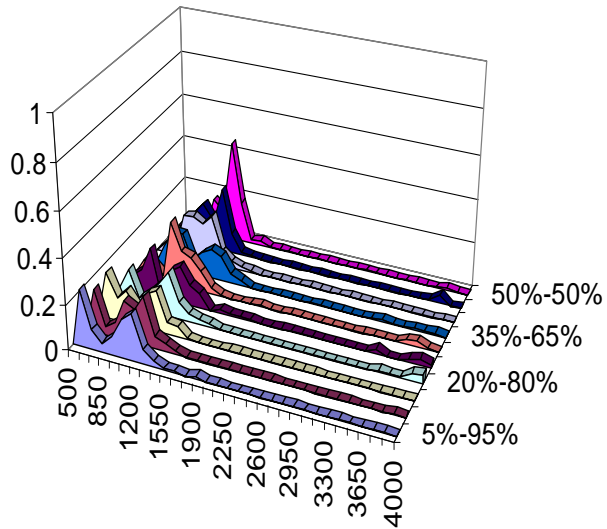


Figure 8c. Fine Grain Open Loop (CPU Bound)

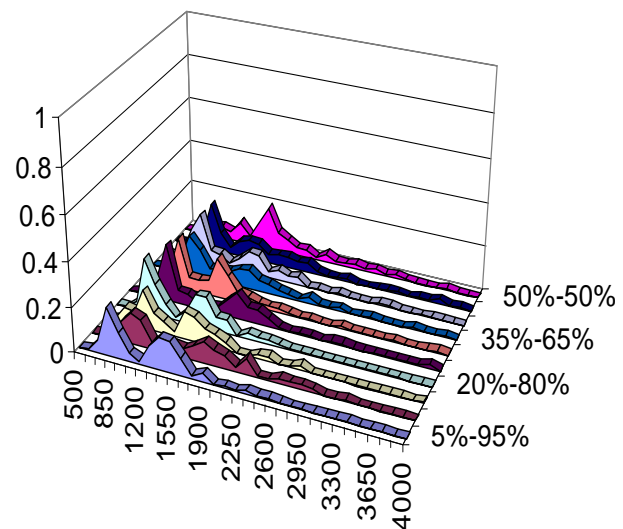


Figure 8d. Fine Grain Open Loop (Mixed CPU/IO)

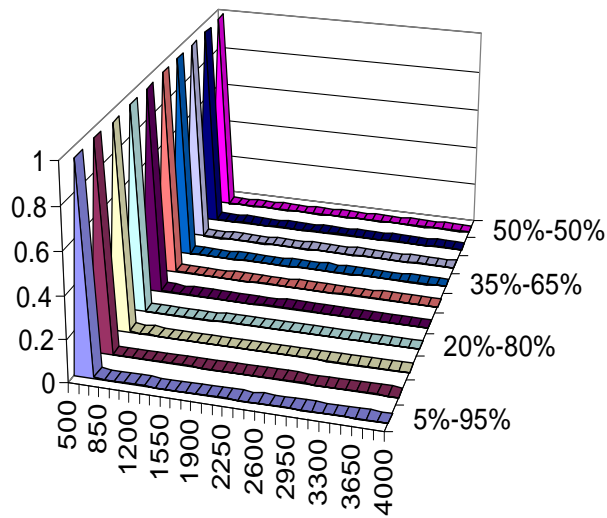


Figure 8e. Closed Loop (CPU Bound)

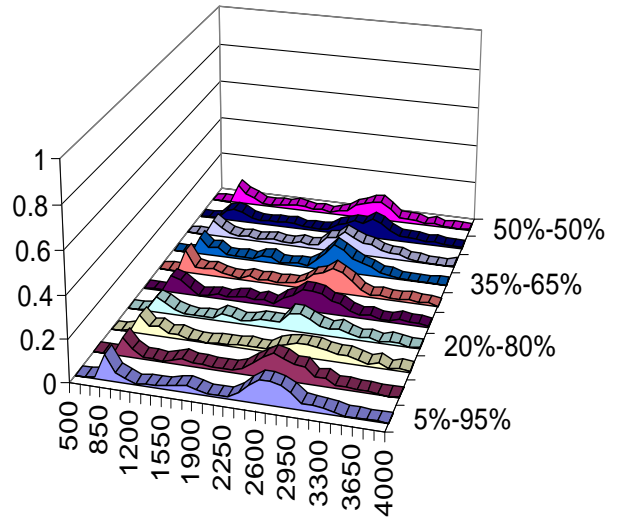


Figure 8f. Closed Loop (Mixed CPU/IO)

Figure 8. Context Switch Times

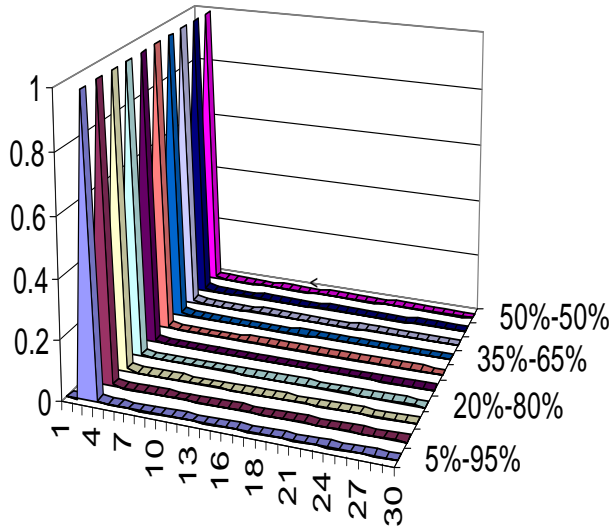


Figure 9a. Coarse Grain Open Loop (CPU Bound)

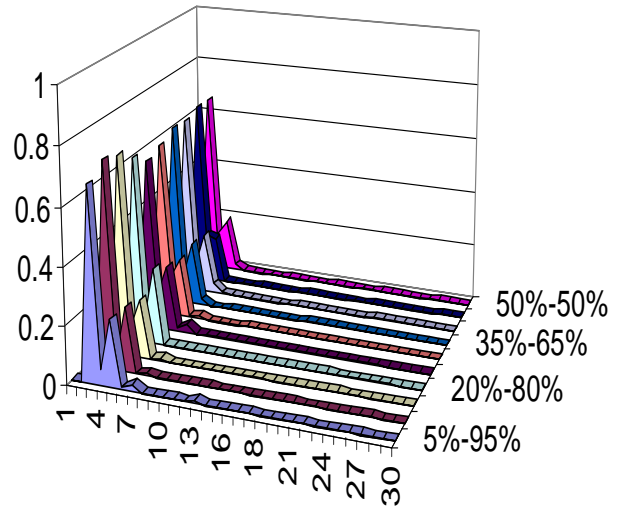


Figure 9b. Coarse Grain Open Loop (Mixed CPU/IO)

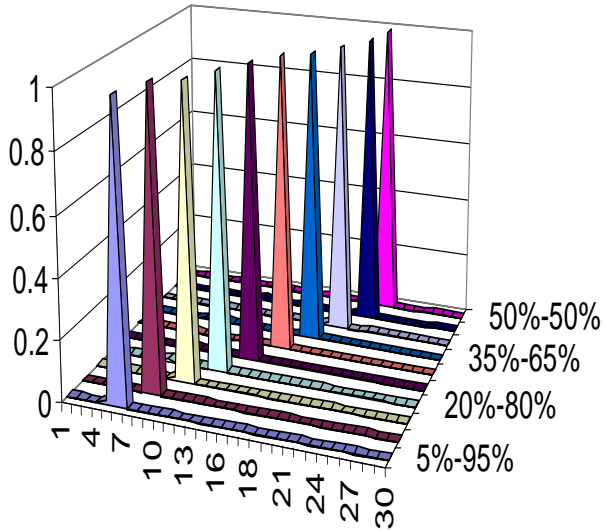


Figure 9c. Fine Grain Open Loop (CPU Bound)

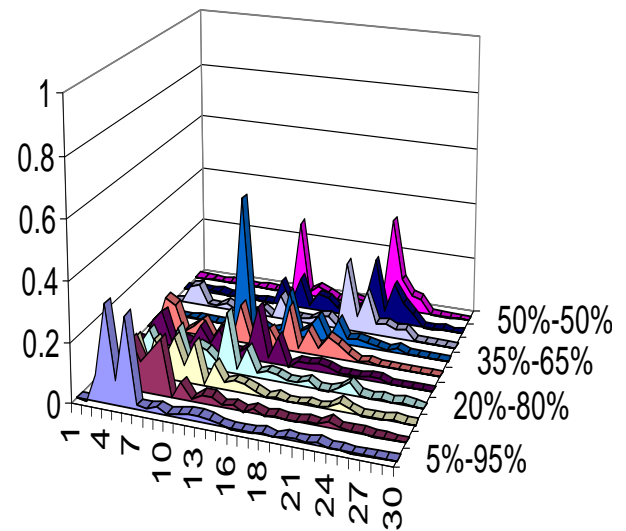


Figure 9d. Fine Grain Open Loop (Mixed CPU/IO)

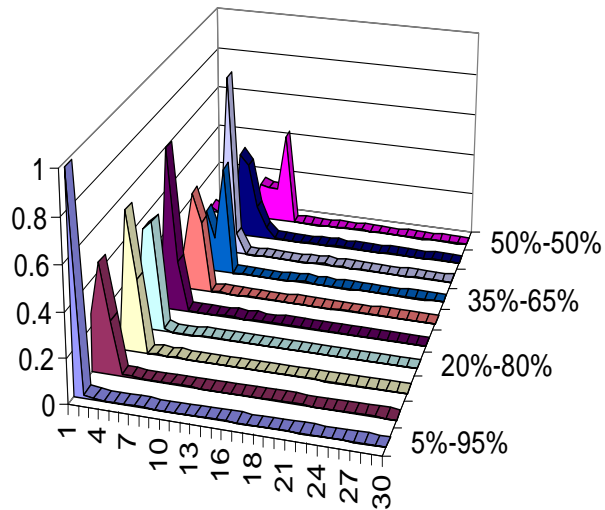


Figure 9e. Closed Loop (CPU Bound)

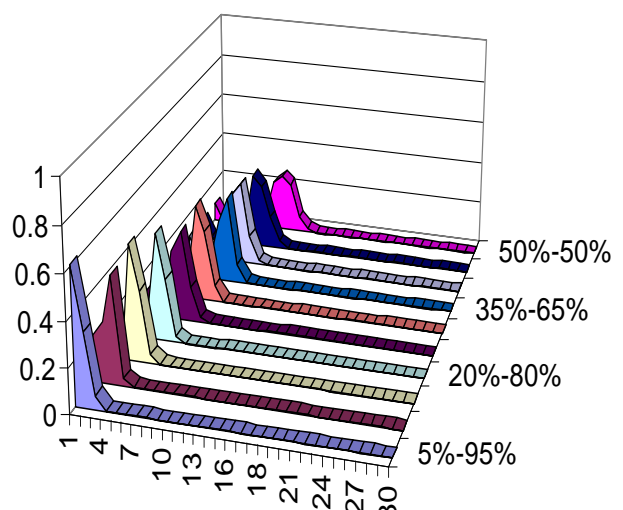


Figure 9f. Closed Loop (Mixed CPU/IO)

Figure 9. Context Switch Rates

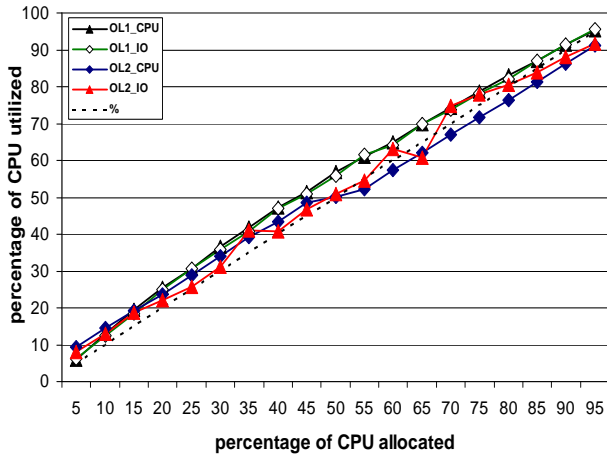


Figure 10a. Open Loop Scheduling Strategies

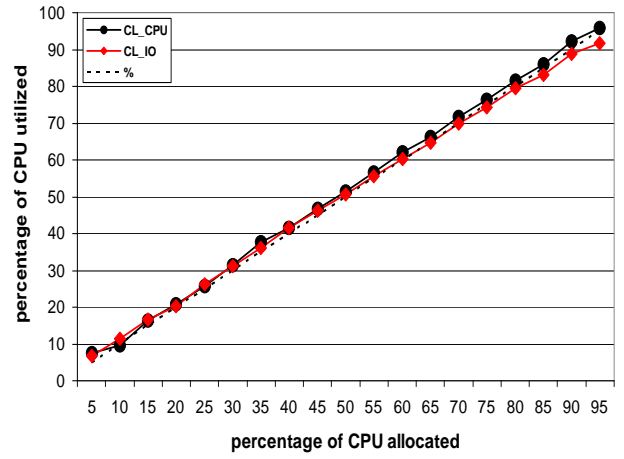


Figure 10b. Closed Loop Scheduling Strategy

Figure 10. CPU Utilization vs. CPU Allocation

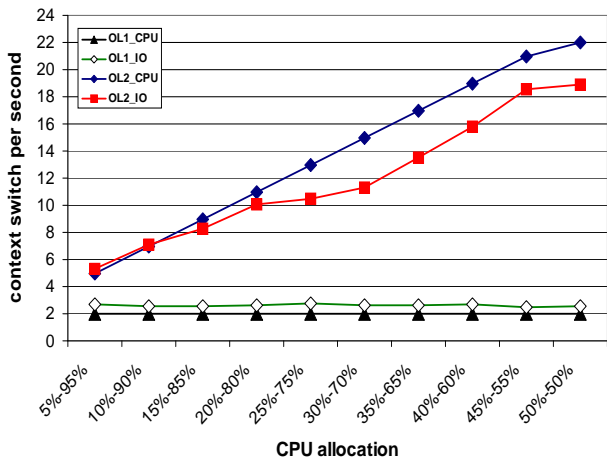


Figure 11a. Open Loop Scheduling Strategies

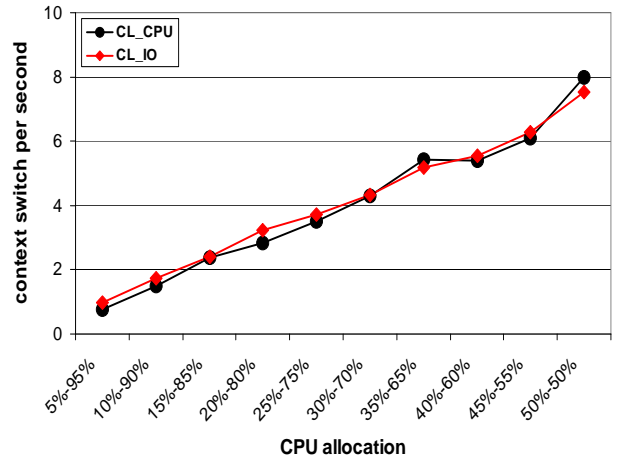


Figure 11b. Closed Loop Scheduling Strategy

Figure 11. Context Switch Rate vs. CPU Allocation

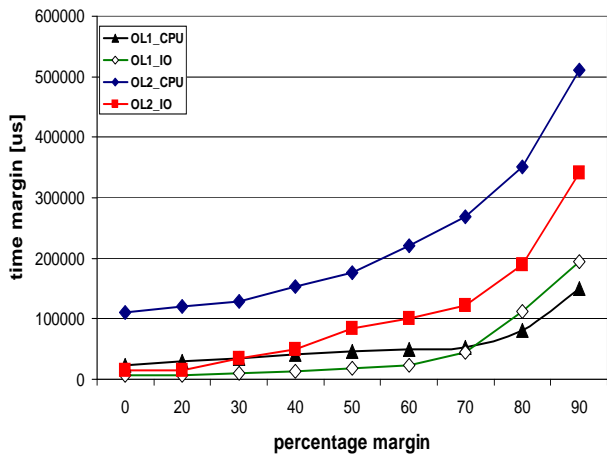


Figure 12a. Open Loop Scheduling Strategies

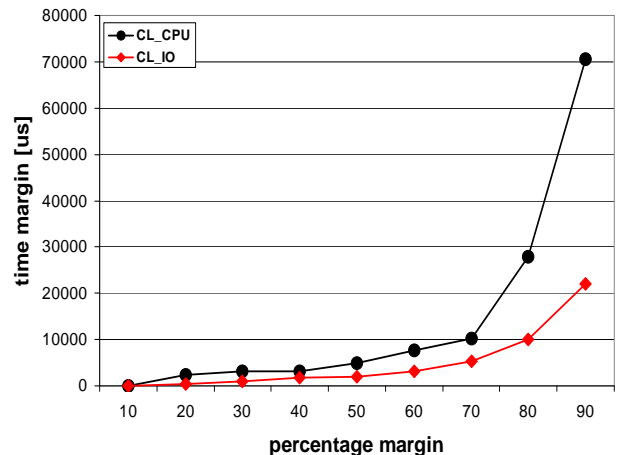


Figure 12b. Closed Loop Scheduling Strategy

Figure 12. Time Margins vs. CPU Allocation

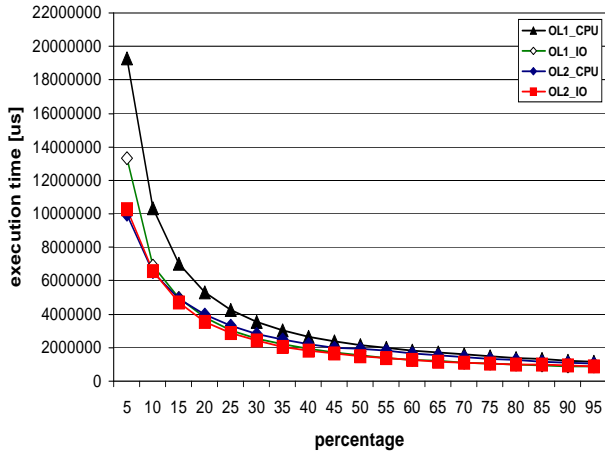


Figure 13a. Open Loop Scheduling Strategies

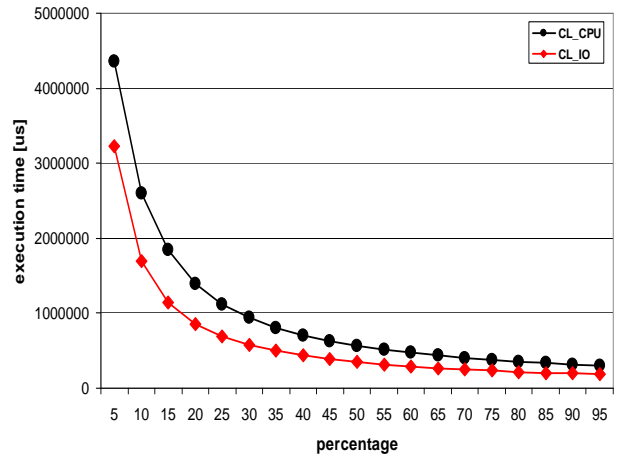


Figure 13b. Closed Loop Scheduling Strategy

Figure 13. Execution Times vs. CPU Allocation

5.2 Analysis of Results

Precision in managing process execution. The figures shown in Section 5.1 confirm expected results: the precision of the scheduling control is mostly accurate in the case of CPU-bound processes and it increases when the CPU allocation increases. In other words, the process' execution time is more predictable when a process gets more CPU cycles. This is valid also in the case of mixed CPU-IO bound processes, although samples are more scattered, and distributions are not tight as the CPU bound case.

- *Open-loop management:* As far as open-loop management is concerned, the second (fine-grain) scheduling strategy is more precise than the first (coarse-grain) one. This is confirmed by comparing distributions shown in Figures 7a, 7b, 7c, 7d and the actual percentage's curves shown in Figure 10a (OL1 stands for Open-Loop, first (coarse-grain) strategy, while OL2 stands for Open-Loop, second (fine-grain) strategy).

This is due to the different decisions used to let a process progress. The first (coarse-grain) Round-Robin algorithm allows a process to progress only after the process ahead of it (in the scheduling queue) consumes its time quantum. The second (fine-grain) Round-Robin algorithm allows processes to progress together, whenever possible. Therefore, the effect of missing an opportunity to accomplish the process' tasks within the scheduling round are more harmful in the coarse-grain strategy, since the process has to wait the whole time quantum of the next process to get a chance to run again. This can result in *multi-modal* distributions of its execution time, as is illustrated in Figures 7a and 7b. When the process behavior changes from the CPU bound to the mixed CPU-IO bound, the scheduling precision decreases. In fact, it is possible that processes might miss the chance to run within the round or to consume the whole quantum's time. Since no feedback is provided, the scheduling strategy becomes less predictable, though still providing reasonably good performance, as is confirmed by the curves in Figure 10a.

- *Closed-loop management:* the closed-loop management approach shows a behavior that is even more precise than the open-loop management approaches, in both CPU bound and mixed CPU-IO bound scenarios (Figures 7e, 7f). However, it produces *multi-modal* time distributions at low CPU allocation levels, so it is more applicable to cases where allocations are more even between processes. This is due in part to the particular behavior adopted by the processes chosen in the experiments. The execution time is computed as the margin between two timestamps at the beginning and at the end of the periodic task. It might happen

that these timestamps are taken in different time slots, resulting in different execution time samples. As the CPU percentage increases, the number of available scheduling slots increases, making such a phenomenon less evident. Thus, the closed loop approach is sensitive to how the feedback data on which it operates is collected. The precision of the closed-loop management is also shown by the diagram in Figure 10b, with a small deviation between the actual percentage curve and the ideal one.

Group Scheduling overhead. Every time a call to the group scheduler was triggered, the time spent by the scheduler to pick and schedule a new process was measured to be in the range of thousands of CPU cycles, which results in a range of *microseconds* on a 2.5GHz machine. Moreover, these times were shown to be predictable across different CPU allocations.

- *Open-loop management:* as illustrated in Figures 8a, 8b, 9a, 9b, and 11a the first (coarse-grain) scheduling strategy’s overhead is generally lower and more predictable than the second (fine-grain) strategy’s overhead across different CPU allocations, in both CPU and mixed CPU-IO bound cases. Specifically, in the CPU bound case the number of context switches remains constant in the first (coarse-grain) strategy, since all of the quanta allocated to a process are scheduled contiguously, and the number of context switches is thus directly proportional to the number of processes. With the second (fine-grain) strategy, however, quanta are interleaved so that a more even division of the CPU among the processes results in fewer contiguous quanta being used by a particular process, and greater overhead due to more context switches between processes.

In our experiments, the scheduler dealt with two processes, with the scheduling round set to 1 second (i.e. the sum of the quanta). The number of context switches per second for the coarse-grain strategy is hence constantly equal to 2 when processes are CPU-bound, and close to 2 when processes are mixed CPU-IO bound. The second (fine-grain) scheduling strategy’s overhead is rather predictable as well, although it increases as the margin between the CPU allocations decreases.

In the case where CPU usage and I/O were interleaved, the context switch overhead was higher for the first (coarse-grain) scheduling strategy and lower for the second (fine-grain) scheduling strategy, but with each strategy the results were similar to those for the CPU-bound case. We attribute the differences in context switch overhead between the CPU-bound and interleaved CPU-IO cases to the phasing of when processes blocked and unblocked from I/O calls relative to the phasing of when one process was scheduled relative to the other by the scheduling strategy. The highest overhead for the second (fine-grain) scheduling strategy was measured when the processes have the same allocation. In fact, they progress fairly since the scheduler switches between them each slot.

- *Closed-loop management:* the overhead of the closed-loop strategy has a predictable behavior across different CPU allocations, and it increases as the margin between the CPU allocations decreases (Figures 8e, 9e, 8f, 9f). Specifically, it is inversely proportional to the sampling period t_s : a smaller sampling period may require more scheduling decisions, and thus likely more context switches. As with the second (fine-grain) closed loop scheduling strategy, the highest context switch overhead is measured when the processes fairly share the CPU, since the scheduler picks group members alternatively and allows them to progress simultaneously. It should be noted that the closed-loop strategy presents less overhead than the fine-grained open-loop strategy, as illustrated in Figures 8e, 8f, 9e, 9f, and 11b.

Tuning the scheduling control to protect/penalize processes. Figures 12a, 12b, 13a, and 13b show the relationships between execution time and CPU allocation for the scheduling strategies we have examined.

- *Open-Loop management:* Figure 13a illustrates the ratio between the time spent to execute a process and the CPU allocation reserved for it. As Figure 13a shows, the scheduling strategies have more similar behaviors as the CPU allocation increases. As the CPU allocation decreases, behaviors become different and the first (coarse-grain) scheduling strategy (i.e. OL1) penalizes processes more than the second (fine-grain) one (OL2). The execution time is halved approximately each step. Figure 12a illustrates the margin between the execution times spent by a group of two processes which are executed concurrently, but with different CPU allocations. The X axis shows the margin between their CPU allocations. As Figure 12a shows, the margin increases as the CPU allocation increases. Margins seen with the fine-grain OL2 strategy are greater than margins experienced with the coarse-grain OL1 strategy, which reflects a more effective enforcement by the fine-grain strategy.
- *Closed-Loop management:* As results in Figures 12b and 13b show, the closed-loop strategy has a behavior which is similar to the open-loop strategies. It should be noted that curves in Figures 12a, 13a, 12b, and 13b cannot be compared since task periods chosen to test open-loop management were different from the ones used to test closed-loop management.

6 Related Work

In this section we describe related work in the following main areas: operating system security frameworks, operating system scheduling, hierarchical organization of scheduling decision functions, and role-based access control and utilization control.

OS security frameworks. SELinux is based on the Flask security architecture [21], which provides policy enforcement and policy decision-making code. The policy decision-making code is encapsulated in a separate component of the operating system called the security server. The Flask security architecture includes an access vector cache (AVC) component that provides caching of access permissions. The policy enforcement code obtains security policy decisions from the security server and AVC, and applies those decisions to assign security labels to processes and objects, and to control process operations based on those security labels. In the LSM-based SELinux security module, the policy enforcement code is implemented in the hook functions, and the policy-independent data types are stored using the security fields in the kernel data structures.

OS scheduling. The Scout operating system [22] introduced the *path* abstraction, which made explicit the notion of the computation's execution environment cross-cutting system layers. In Scout, paths compose segments of the computation with mechanisms that manage their progress. Flexible scheduling decision functions have been realized in Scout, such as the Best-Effort Real-Time (BERT) [23] algorithm. Our group scheduling approach generalizes the notion of path scheduling to a wider range of platforms, and also shifts the path abstraction from the execution environment model to the scheduling model.

Hierarchical scheduling. Goyal, Guo, and Vin [24] and Regehr and Stankovic [25] have extended the notion of a scheduling decision function to include *composition* of scheduling decision functions within a *hierarchical scheduling* framework. Regehr, *et al.*, have applied hierarchical scheduling to cross-cut alternative execution environments so that analysis of both concurrency and schedulability can be done together [26]. Both hierarchical scheduling and the BERT approach in the Scout operating system compose scheduling

decision functions across system layers. The BERT algorithm for slack stealing in fair-queuing scheduling decision functions can be realized in the more general hierarchical scheduling model. Since group scheduling extends the hierarchical scheduling model, it also extends the BERT approach.

Feedback control scheduling. Feedback control has been used in a variety of scheduling techniques. Applications of these techniques have included web servers [27], ORBs [28], real-time image transmission [29]. When combined with hybrid control techniques [30], feedback scheduling approaches offer significant potential for extension of the closed loop scheduling strategy described in this paper, to incorporate a wide range of security aspects.

Role based access and usage control. Role-based access control (RBAC) [31, 32, 33] allows security policies to be applied flexibly and efficiently to *roles* (which tend to be relatively stable allocations of authority for a system), instead of to individuals (who may change roles or enter or depart from an organization more frequently). Role based access control thus can ease the complexity of administering security access policies by eliminating gratuitous variability in the assignment of access permissions. Sandhu, et al., have developed extensions to RBAC to support *usage control* (UCON) [34] which provides role-based administration of computations behavior. Our approach extends the ideas in UCON by integrating explicit scheduling of *when* computations are executed.

7 Conclusions and Future Work

In this paper we have considered the specific problem of CPU focused denial of service attacks, and in doing so have motivated the more general problem of *execution security* as a crucial issue for time-sensitive applications, particularly as modern computing systems are increasingly ubiquitous, inter-connected, and open.

Summary of contributions. This paper has presented three contributions to the state of the art in protection of the execution of system activities from accidental or adversarial interference. First, in Section 1 we have explained how limitations of current approaches make it difficult to offer sufficiently rigorous and fine-grained assurances of protection for system computations, in the face of CPU-focused denial of service attacks. Second, in Section 3 we have presented a novel solution approach in which fine-grained scheduling decision functions [3, 2, 1] such as those described in Section 4 are integrated with system call hooks from the Security Enhanced Linux (SELinux) framework [4, 17] within the Linux 2.6 kernel. Third, our empirical evaluations in Section 5 have demonstrated the efficacy of our approach in controlling the CPU utilization of competing greedy computations that are either completely CPU bound, or that interleave I/O and CPU access, across a range of relative allocations of the CPU.

Future work. Our near-term research agenda will focus on several additional open problems related to the work presented in this paper: (1) development of a complementary audit and forensics framework in which execution control decisions are recorded and correlated with a rigorous fine-grained accounting of system behavior; (2) efficient end-to-end configuration and on-line enforcement protocols in middleware, and formalization of composition methods that maintain execution control assurances when application components and subsystems are integrated; and (3) integration of our execution control approach with other security techniques such as role-based access control.

We will develop an expanded set of techniques for configuring and checking security constraints when different application components, infrastructure layers, and separate endsystems are composed within a local endsystem. We will also extend this local composition model to include composition of control for application assemblies that are distributed across multiple endsystems.

Finally, we will study how to integrate our execution security approach with existing role-based access and usage control approaches, to provide an integrated role-based approach to both data and execution security. To do this we will examine how hierarchically organized groups used in both the role based access control and group scheduling models can be integrated into a combined model for managing access permissions, and resource admission and execution control policies. We will also examine the implications of enforcement of constraints that cross-cut the access, admission, and execution control dimensions, on the safety and liveness properties of the overall system. Our goal will be to develop a practical and general model for role-based decision functions that can express and compose (1) simple and efficient Boolean decision functions such as access control decisions based on role attributes, (2) more complex predicates such as admission decisions based on resource availability, and (3) longer-running control functions to monitor and steer progress of executing computations that are granted both access and admission.

8 Acknowledgements

We wish to thank Ravi Sandhu for valuable discussions of the relationships between access control and admission and execution control for time-and-security-critical systems. His work on role-based access control and usage control inspired our investigation of how scheduling and admission control techniques from the real-time systems research field could be applied to problems of execution security. We also wish to thank Erik Andersen who helped with issues related to our initial SELinux setup and the first steps involving interaction between SELinux and Group Scheduling.

References

- [1] M. Frisbie, “A unified scheduling model for precise computation control,” Master’s thesis, University of Kansas, June 2004.
- [2] M. Frisbie, D. Niehaus, V. Subramonian, and C. Gill, “Group scheduling in systems software,” in *Workshop on Parallel and Distributed Real-Time Systems*, (Santa Fe, NM), apr 2004.
- [3] T. Aswathanarayana, V. Subramonian, D. Niehaus, and C. Gill, “Design and performance of configurable endsystem scheduling mechanisms,” in *Proceedings of 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2005.
- [4] B. McCarty, *SELINUX: NSA’s Open Source Security Enhanced Linux*. O’Reilly, October 2004.
- [5] Douglas Niehaus, *et al.*, “Kansas University Real-Time (KURT) Linux.” www.ittc.ukans.edu/kurt/, 2004.
- [6] D. Niehaus, “Improving support for multimedia system experimentation and deployment,” in *Workshop on Parallel and Distributed Real-Time Systems*, (San Juan, Puerto Rico), Apr. 1999. Also appears in Springer Lecture Notes in Computer Science 1586, Parallel and Distributed Processing, ISBN 3–540–65831–9, pp 454–465.
- [7] B. Buchanan, D. Niehaus, D. Dhandapani, R. Menon, S. Sheth, Y. Wijata, and S. House, “The data stream kernel interface,” Tech. Rep. ITTC-FY98-TR11510-04, Information and Telecommunication Technology Center, University of Kansas, 1998.

- [8] C. Shields, ““what do we mean by network denial of service?”,” in *IEEE Workshop on Information Assurance and Security*, IEEE, jun 2002.
- [9] Tim Yardley, “explanation and code for streamc issues.” www.security-express.com/archives/bugtraq/2000-01/0283.html , Jan. 2000.
- [10] sl aka spunone@FAZED.NET, “various *lame* DoS attacks.” cert.uni-stuttgart.de/archive/bugtraq/1998/11/msg00107.html , Nov. 1998.
- [11] I. A. Finlay, “Apache mod dav module vulnerable to dos.” <http://www.kb.cert.org/vuls/id/4061> 21 , Sept. 2002.
- [12] D. Keromyliis, V. Misra, and D. Rubenstein, “SOS: An Architecture For Mitigating DDoS Attacks,” *IEEE Journal on Selected Areas in Communications*, pp. 176–187, jan 2004.
- [13] B. A. Kuperman, C. E. Brodley, H. Ozdoganoglu, T. N. Vijaykumar, and A. Jalote, “Detection and Prevention of Stack Buffer Overflow Attacks,” *Communications of the ACM*, vol. 48, pp. 51–56, Nov. 2005.
- [14] S. Crosby and D. Wallach, “Denial of service via algorithmic complexity attacks,” in *12th USENIX Security Symposium*, USENIX, aug 2003.
- [15] Wikipedia, “Fork bomb.” en.wikipedia.org/wiki/Fork_bomb .
- [16] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus, “A Firm Real-Time System Implementation Using Commercial Off-The-Shelf Hardware and Free Software,” in *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium*, (Denver, CO), IEEE, June 1998.
- [17] Stephen Smalley and Chris Vance and Wayne Salamon, “Implementing SELinux as a Linux Security Module.” <http://www.nsa.gov/selinux/papers/module/t1.html> .
- [18] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis, “Weighted round-robin cell multiplexing in a general purpose ATM switch chip,” *IEEE Journal on Selected Areas in Communications*, vol. 9, pp. 1265–1279, oct 1999.
- [19] Wikipedia, “Weighted round robin.” en.wikipedia.org/wiki/Weighted_round-robin_scheduling .
- [20] “Linux Security Modules.” lsm.inmunix.org/lsm_about.html .
- [21] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau, ““the flask security architecture: System support for diverse security policies”,” in *8th USENIX Security Symposium*, USENIX, aug 1999.
- [22] D. Mosberger and L. L. Peterson, “Making Paths Explicit in the Scout Operating System,” in *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, USENIX Association, Oct. 1996.
- [23] A. Bavier, L. Peterson, and D. Mosberger, “BERT: A Scheduler for Best Effort and Realtime Tasks,” Tech. Rep. TR-602-99, Princeton University, 1999.

- [24] Goyal, Guo, and Vin, "A Hierarchical CPU Scheduler for Multimedia Operating Systems," in *2nd Symposium on Operating Systems Design and Implementation*, USENIX, Oct. 1996.
- [25] Regehr and Stankovic, "HLS: A Framework for Composing Soft Real-Time Schedulers," in *22nd IEEE Real-Time Systems Symposium*, (London, UK), Dec. 2001.
- [26] Regehr, Reid, Webb, Parker, and Lepreau, "Evolving real-time systems using hierarchical scheduling and concurrency analysis," in *24th IEEE Real-Time Systems Symposium*, (Cancun, Mexico), Dec. 2003.
- [27] R. Zhang, C. Lu, T. Abdelzaher, and J. Stankovic, "Controlware: A Middleware Architecture for Feedback Control of Software Performance," in *Proceedings of the International Conference on Distributed Systems 2002*, July 2002.
- [28] C. Lu, X. Wang, and C. Gill, "Feedback Control Real-Time Scheduling in ORB Middleware," in *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, (Washington, DC), IEEE, May 2003.
- [29] X. Wang, H.-M. Huang, V. Subramonian, C. Lu, and C. Gill, "CAMRIT: Control-based Adaptive Middleware for Real-time Image Transmission," in *Proc. of the 10th IEEE Real-Time and Embedded Tech. and Applications Symp. (RTAS)*, (Toronto, Canada), May 2004.
- [30] X. Koutsoukos, R. Tekumalla, B. Natarajan, and C. Lu, "Hybrid Supervisory Control of Real-Time Systems," in *11th IEEE Real-Time and Embedded Technology and Applications Symposium*, (San Francisco, California), Mar. 2005.
- [31] R. Sandhu, "Rationale for the RBAC96 family of access control models," in *Proceedings of the 1st ACM Workshop on Role-Based Access Control*, ACM, 1997.
- [32] R. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *IEEE Computer*, vol. 29, pp. 38–47, February 1996.
- [33] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, "Proposed nist standard for role-based access control," *ACM Transactions on Information and System Security*, vol. 4, pp. 224–274, August 2001.
- [34] J. Park and R. Sandhu, "The UCON_{ABC} usage control model," *ACM Transactions on Information and System Security*, vol. 7, pp. 128–174, February 2004.