2006-34

The Design, Modeling, and Implementation of Group Scheduling for Isolation of Computations from Adversarial Interference

Authors: Tidwell, Terry; Watkins, Noah; Subramonian, Venkita; Niehaus, Douglas; Gill, Christopher; Migliaccio, Armando

June 1, 2006

Corresponding Author: cdgill@cse.wustl.edu

Abstract: To isolate computations from denial of service (DoS) attacks and other forms of adversarial interference, it is necessary to constrain the effects of interactions among computations. This paper makes four contributions to research on isolation of computations from adversarial interference: (1) it describes the design and implementation of a kernel level scheduling policy to control the effects of adversarial attacks on computations™ execution; (2) it presents formal models of the system components that are involved in a representative DoS attack scenario; (3) it shows how model checking can be used to analyze that example scenario, under default Linux scheduling semantics and under our scheduling policy design; and (4) it presents empirical studies we have conducted to validate our scheduling policy implementation. Our results show that, with careful design, scheduling and detailed monitoring of computations™ behavior can be combined effectively to mitigate interference of attacks with computations™ execution.

The Design, Modeling, and Implementation of Group Scheduling for Isolation of Computations from Adversarial Interference

Terry Tidwell[•], Noah Watkins[◦], Venkita Subramonian[•], Douglas Niehaus[◦∗], Christopher Gill[•†], Armando Migliaccio[‡]

[◦]Department of Electrical Engineering and Computer Science, University of Kansas, Lawrence, KS, USA

[•]Department of Computer Science and Engineering, Washington University, St. Louis, MO, USA

[‡]Dipartimento di Informatica e Sistemistica, Universita degli Studi di Napoli Federico II, Italy

Abstract

To isolate computations from denial of service (DoS) attacks and other forms of adversarial interference, it is necessary to constrain the effects of interactions among computations. This paper makes four contributions to research on isolation of computations from adversarial interference: (1) it describes the design and implementation of a kernel-level scheduling policy to control the effects of adversarial attacks on computations' execution; (2) it presents formal models of the system components that are involved in a representative DoS attack scenario; (3) it shows how model checking can be used to analyze that example scenario, under default Linux scheduling semantics and under our scheduling policy design; and (4) it presents empirical studies we have conducted to validate our scheduling policy implementation. Our results show that, with careful design, scheduling and detailed monitoring of computations' behavior can be combined effectively to mitigate interference of attacks with computations' execution.

1. Introduction

Society is increasingly dependent on complex mission-critical systems such as supervisory control and data access (SCADA) systems for power grid management [1], industrial control systems for automated manufacturing [2], and medical device systems for patient care [3]. Because attacks against these systems may cause their failure, and because of the safety, health, and economic concerns that such failures would entail, these systems constitute *critical infrastructures* of vital interest, whose survivable operation must be protected even in the face of malicious attacks.

Recent trends toward tele-operation and monitoring of these systems, and toward connecting them with non-critical systems that can be accessed through public networks, have exacerbated their risk of failure due to adversarial attack. The ability of an attacker to interact even indirectly with the system may allow the attacker to *interfere* with the system's correct operation. We are concerned specifically with adversarial interference involving sequences of interactions over time that can cause a system to violate its behavioral constraints.

Adversarial interference. The central problem the research presented in this paper addresses, is how to design systems whose computational behavior is *survivable* in that it does not violate specified constraints even in the face of adversarial attack. While techniques such as network packet filtering [4, 5], have been developed to deter malicious use of networks, less attention has been paid to how to handle exploitation of other interaction channels by an adversary. The ability of an attacker to interfere with the execution of key system *computations* (e.g., those running control algorithms) that are responsible for ensuring the system's correct operation is frequently not addressed adequately in many systems, particularly those developed with commercially available hardware and software. For example, most common off the shelf (COTS) operating systems are designed for use as "black box" abstractions, with few opportunities for the system developer to refine the semantics of crucial features such as thread scheduling and interrupt handling.

This approach often forces the system designer to use indirect mechanisms to achieve desired application execution semantics, often increasing the application's vulnerability to interference from other computations subverted by an attacker. For example, semaphores are often used as an *indirect* mechanism for implementing specialized scheduling semantics when *direct* manipulation of the OS API features available (usually priorities) does not have the desired semantics. This approach introduces the system's semaphore support mechanisms as an interaction channel that may be exploited by an attacker.

∗ Research supported in part by by NSF EHS contract CCR-0311599.
† Research supported in part by NSF CAREER Award CCF-0448562

Group scheduling framework. Our previous research on precise scheduling in real-time systems [6] has resulted in a group scheduling framework that is also suitable for specifying and enforcing isolation among computations. As we describe in Section 3, we have developed new techniques within the group scheduling framework to prevent types of adversarial interference to which COTS systems are vulnerable due to their inability to control computation behavior precisely. The essential features of the group scheduling model are: (1) the ability to *group* application components with system components supporting them; (2) the ability to associate customized scheduling decision functions with groups; (3) the ability to compose groups to form a unified system-wide policy for resource management; (4) the ability to enforce the specified resource use semantics as defined in the composed group policy structure; and (5) the ability to measure system behavior at the same level of detail at which the application specifies its semantics, and to use measured performance statistics as feedback to the scheduling decision functions.

Model and system co-design. System designers, especially in the context of applications with significant behavioral constraints, benefit from the ability to model the semantics of the application, since design verification and implementation validation are crucial to ensure computations' behavioral constraints are satisfied, especially in the face of adversarial attack. However, an application's behavioral semantics depends in part on how the application is supported by various operating system services. In the driving example used for this paper, which we discuss in Section 2.3, these services include the scheduler and any OS-level components such as hard IRQs and soft IRQs that process network packets. A model for each computation must thus incorporate models of the behavior of all system components affecting the application behaviors of interest.

Careful co-design of models and implementations is motivated by trade-offs among several important issues, including: model fidelity, model tractability, enforcement precision, implementation complexity and performance. The ease and accuracy with which the system can be modeled depends on the semantics of the system implementation. Our experience with the driving problem discussed in this paper has shown that precise measurements of the system can inform modeling choices significantly, and modeling complexity and fidelity concerns can influence system implementation choices strongly. Our goal is to develop system implementations capable of meeting their behavioral constraints, for which we can also build tractable models of sufficient fidelity that satisfaction of behavioral constraints can also be verified.

Paper outline. Section 2 introduces our system and attack models, and presents a denial of service attack scenario that we use as a driving example. Section 3 describes the design and implementation of group scheduling policies and mechanisms for isolation of computations from adversarial interference. Section 4 presents timed automata models we have developed for relevant system components. Section 5 presents an empirical validation of our implementation that compares modeled behavior to that of the actual system. Section 6 describes other work related to the research presented in this paper, and Section 7 offers concluding remarks and summarizes future work.

2. Attack Model and Scenario

While interference among components of a computation is entirely possible due to poor design or to operating conditions that exceed design specifications, in this section we focus on the specific problem of *adversarial* interference, and in particular on the ways in which an attacker can or cannot use subverted computations to interfere with other computations that have not been subverted. To examine in detail the problems that can arise with these approaches in the face of adversarial attack, Section 2.1 describes a system model that defines more precisely how computations can interact and potentially interfere with each other's execution. Section 2.2 describes an attack model based on that system model. Section 2.3 then describes an attack scenario that we use as a driving example throughout this paper.

2.1. System Model

We first describe four basic elements of our system model: (1) **events** that denote relevant asynchronous changes in the state of the system (e.g., a thread blocking on an operating system call); (2) **actions** that modify the state of the system (e.g., making a function call in user-space, or changing which thread is currently executing on the CPU) – we assume that each action takes a non-zero time interval to complete and that an action's initiation and completion may be tagged by events if they are relevant; (3) **decisions** that determine which actions are taken and when; and (4) **contexts**, which record time-indexed information about all of the other elements of the system model, and are used to increase the specificity of actions and decisions (e.g., to *which* thread a decision or action pertains).

We also describe six higher level elements of our system model, which build on the four basic elements, and which serve as the basis for describing the attack models and the system services needed to address those attack models in this paper: (5) **application-level programs** such as control, video processing, and data fusion software – these programs are developed separately from the OS kernel and its associated libraries, execute entirely in user-space, and make decisions and perform actions during their execution; (6) **system**

components which provide services to application-level programs and respond to events such as the arrival of a network packet – some system components (e.g., tasklets, IRQ handlers, soft IRQs and kernel-level threads) run within the kernel, while others (e.g., application and middleware threads) run in user-space; (7) **computations** which represent the composed semantics of each program and the system components that run on its behalf, and which thus capture the structure of *dependencies* among application-level programs and system components; (8) **interaction channels** which are resources and/or system components shared among computations in ways that can result in the actions of one computation having consequences for the execution of another computation; (9) **behavioral constraints** which are specified predicates over the order and timing of events within a computation (e.g., *when* a result is returned from a function call) that determine the correctness of its execution; and (10) **interference** which is a causal sequence of actions and events involving one or more computations that leads to the violation of a behavioral constraint.

2.2. Attack Model

We assume an attack model with the following features: (1) each program runs in a **separate, protected process address space**; (2) in general, any individual program can be **subverted** by an attacker that can gain access to it (e.g., through a buffer overflow, "Trojan horse" program, or virus); (3) any subverted program can take **arbitrary actions**; (4) system components in the kernel cannot be subverted, so the semantics of their decisions and actions remain **consistent** across all computations – even ones executing subverted programs; and (5) system components that execute in user space can be subverted by a subverted program, but only the **copy** of the component that is run within the subverted program's address space is subverted, which does not subvert other copies of the component running within other programs' address spaces.

The first attack model feature limits the scope of an individual subversion act to the process subverted. Making one program execute in arbitrary ways does not imply the ability to make another program behave in arbitrary ways, since for the purposes of this research we assume that the address space of each process is protected from modification by another process except through explicitly created shared data areas. Thus, an attacker wishing to subvert multiple processes must subvert each one individually.

The second and third features of the attack model mean that in theory an attacker can cause any program in the system to behave in arbitrary ways, though in practice this ability is significantly constrained by (1) the routes through which the attacker can subvert a program, (2) process and endsystem boundaries between computations, and (3) the limited set of interaction channels through which an attacker can influence other computations in the system. This limitation is particularly relevant to many complex engineered systems, since the only programs an attacker can subvert directly are often those on vulnerable endsystems (e.g., in a business information system that is connected to a public network) rather those inside the engineered system itself (e.g., within a SCADA system to which the business information system is connected). Many systems try to address the implications of a subverted program's ability to take arbitrary actions by correcting those actions through fault tolerance, byzantine agreement or other approaches that are outside the proposed research. The work proposed here concentrates on limiting the ability of a subverted computation to affect others, not on correcting the fact of subversion. The subverted process can, however, *tell lies to schedulers and other system components*.

The fourth and fifth features of our attack model mean that while an attacker can make the entire user-space portion of a computation perform arbitrary actions, kernel-level resources and system components will maintain their defined semantics. We make this assumption, in part, because once a thread executing in kernel mode and with access to the kernel address space is executing subverted code there is little that can be done to constrain the attacker's freedom of action, and we do not attempt to address those types of attacks in this paper. This distinction has significant implications for the design of policies and mechanisms for isolation of computations from interference by other computations, and for the inherent limits on isolation that are achievable in each particular system.

2.3. Denial of Service Attack Scenario

Examples of the kinds of adversarial interference our approach is designed to address include denial of service attacks such as: *process shutdown attacks* in which a subverted program attempts to shut down other processes, e.g., by sending them kill signals; *fork bomb attacks* in which a subverted program recursively duplicates itself to occupy, and thus deny access of other computations to, system resources such as process descriptor tables and the CPU; and *request flood attacks* in which a subverted program exploits local and network inter-process communication mechanisms to deny other computations access to network and other resources on the local endsystem, on remote endsystems, or both. These kinds of attacks fit within our attack model and help to illustrate how interference between computations can occur in practice.

As the driving example for this paper, we focus on a *request flood attack* involving interaction channels that span multiple system components. In such an attack, a subverted program exploits inter-process communication mechanisms to send numerous application-level requests (e.g., for method invocations) to another *target* computation and in doing so interferes with the execution of another *victim* computation.

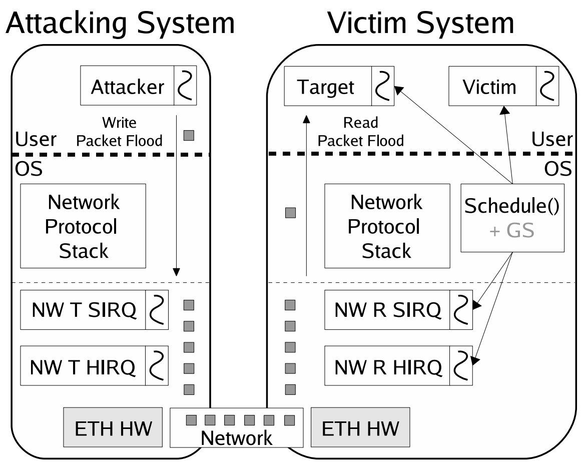

Figure 1. Request Flood Attack Scenario

Figure 1 illustrates such an attack. On an endsystem receiving request packets from the subverted program, a hard IRQ handler executes to notify the system of a packet's arrival, and in Linux a soft IRQ system component must then perform an irreducible amount of processing to classify the request packets. Since in many COTS operating systems, including Linux, system components servicing interrupts are controlled under "as fast as possible" semantics, this can result in excessive use of the CPU by packet processing and denial of CPU access to other computations. Once the packets are classified, the kernel-level scheduler selects the processes to whom the packets are being sent. Here as well, the default semantics of demand-driven scheduling in COTS operating systems can lead to excessive occupation of the CPU by the computation to which the requests are targeted, leading to interference by that computation with still other computations in the same endsystem. In the request flood example, therefore, an attacker may be able to produce a constraint violation in either the computation servicing requests or in other computations by causing interference through the CPU interaction channel. This example is of particular interest because the victim computation is attacked directly via an interaction channel that it is using (the CPU), but that attack is enabled by another interaction channel which the victim computation does not use (the network interface) but to which its execution semantics is nonetheless *indirectly coupled*.

3. Scheduling Design and Implementation

The example attacks described in Section 2.3 include several ways an attacker can cause interaction, and potential interference, with victim computations. They also illustrate an important fact: *a large number of possible interactions pass through a small set of system components*. This is the fundamental insight on which our approach is based: by using kernel-level scheduling and performance monitoring to exert precise control over a core set of system components, we can strongly constrain the ability of attackers to interfere with victims under a large number of potential attack scenarios.

Specifically, in our driving example precise execution control over the soft IRQ and hard IRQ system components supporting the threads serving requests enables us to constrain the ability of a request flood attack to interfere with the victim computation through the CPU interaction channel. Group scheduling is the mechanism through which we exercise precise control over all computational components in KURT-Linux, at both the application and OS level. The precision of group scheduling control is matched by the precision with which we can measure system behavior using the Datastreams subsystem of KURT-Linux.

Previous research has shown the ability of group scheduling to ensure timeliness in distributed real-time and embedded systems [6, 7]. Direct expression of scheduling semantics for computations under group scheduling has three major benefits: (1) it relieves developers of the error-prone task of using indirect mechanisms such as semaphores and priority manipulations which may also increase the application's vulnerability to adversarial attacks by increasing the number of interaction channels; (2) it more effectively supports analysis using formal models by making explicit how scheduling policies interact with system components; and (3) it enables higher fidelity and more efficient design and verification of scheduling policies for constraining interference.

3.1. Default Scheduling Semantics Consequences

Most operating systems, including Linux, support an application programming environment through which programs can interact with a rich set of system-level components. Interrupt handlers are an obvious example, but other relevant components in Linux include several soft IRQs and tasklets. An important implication of this common OS architecture is that some portion of the work specified by an application program is actually performed by OS components. In our driving example, the request packets flooding the victim system are processed by two system components: the hard IRQ handler for the Ethernet device, and the network receive soft IRQ handler.

The crucial point is that, under the default Linux scheduling semantics, execution of both of these OS components is given preference over that of application threads. Moreover, no constraint is placed on the CPU cycles that can be consumed by these components. Measurements we discuss in Section 5 showed that a single machine generating requests can cause the network receive soft IRQ to consume between 50 and 60% of the victim system's CPU cycles, and two machines generating requests can raise that to in excess of 95%. The computations sharing the victim system with the target thread are reduced to sharing whatever limited CPU cycles remain under the default Linux priority scheduling semantics. Even raising the priority of the victim application is of limited utility because, in general, giving a user process a higher priority than the packet processing hard or soft IRQ components is undesirable. Group scheduling provides a framework within which better solutions to this problem can be designed, by creating a constraint on the percentage of the CPU which can be consumed by the group of application and system components affected by the request flood attack, without changing the original prioritization semantics among the group members.

3.2. Precise Computation Component Control

Precise computation control has been a central theme of KURT-Linux from the very beginning. The need for a general framework for controlling groups of application threads motivated the creation of Group Scheduling [7], and the desire for precise control over every aspect of the system affecting CPU use motivated its extension to context-borrowing computations, hard IRQs, soft IRQs and tasklets, in Linux 2.4 [8]. Other work has shown the ability to control groups of application threads and OS computation components to reduce the instrumentation effect of Datastreams [9], and to implement application semantics directly in the scheduling framework for complex multi-threaded multi-pipelined video processing applications [6].

This previous research has produced the group scheduling framework in which we can express directly the precise computation control semantics that our approach to protecting the victim thread in the request flood attack scenario requires. The precise control of of OS computation components has been helped by recent work on the "RT Patch" for Linux 2.6 [10], which significantly improves preemptability in the OS and eliminates almost all context borrowing OS components by creating kernel threads in whose context all hard IRQ and soft IRQ handlers are executed. Given its significant improvement in performance, we have used the RT Patch as the foundation for KURT-Linux in the 2.6 kernel. The group scheduling framework in this context can control essentially every system activity with fine precision, by controlling execution of hard and soft IRQ threads.

3.3. Resource Isolation under Group Scheduling

The precise control of all computation components under KURT-Linux group scheduling makes a wide range of execution semantics possible. Intervals during which a thread uses a particular resource can be tracked using the fine-grain system time standard, e.g. the 64-bit Time Stamp Counter (TSC) on the x86 which increments at CPU clock speed. High resolution timers, such as those provided by KURT-Linux in the 2.4 kernel or by the RT Patch in the 2.6 kernel, can be used to schedule preemption of resource use intervals for resources permitting preemption. For those that do not, resource use control and accounting points can be implemented within scheduling decision functions or inserted in resource acquire and release routines. We used a CPU share *execution opportunity* semantics to control simulated video processing pipelines in [6]. For the request flood attack scenario considered in this paper, we chose a CPU share *usage* semantics to control computations related to receiving network packets: (1) the network receive hard IRQ handler, (2) the network receive soft IRQ handler, and (3) the target thread. CPU use by each execution interval of a thread under control of the scheduling decision function (SDF) we have designed for that scenario (which we discuss in Section 3.4) is measured within schedule(), and the SDF tracks CPU use within accounting windows of configurable length. CPU use by the network receive soft IRQ is non-preemptable under our SDF, which matches its default Linux execution semantics. However, further analysis and modification of the handler could make it possible to use preemption semantics, as future work.

3.4. Scheduling Decision Function Design

The SDFs we used to address the request flood attack scenario that was described in Section 2.3 (and illustrated in Figure 1) are organized hierarchically into a single System Scheduling Decision Function (SSDF) whose structure is illustrated in Figure 2. This structure is essentially a decision tree evaluated by invocation at the root, which returns a decision after recursive evaluation of individual SDFs associated with the nested groups. This structure controls *almost all* CPU use in the system. The only exceptions under the RT Patch are the hard IRQ *service routines* whose execution is invoked directly at the hardware level by interrupt occurrence. However, their executions are extremely short, generally running only the few instructions necessary to enable the associated hard IRQ *service thread*.

The root of the SSDF in Figure 2 is the *schedule()* function implementing the Linux scheduler, which first evaluates the SDF of the subordinate group on its left branch. If this decision tree chooses a thread, then *schedule()* jumps directly to its context switch section. If not, it consults the right branch, which is the default Linux thread scheduler. The top node of the left sub-tree uses a simple sequential SDF (SEQ), which checks its branches from left to right. It checks to see if any hard IRQ threads are able to run,

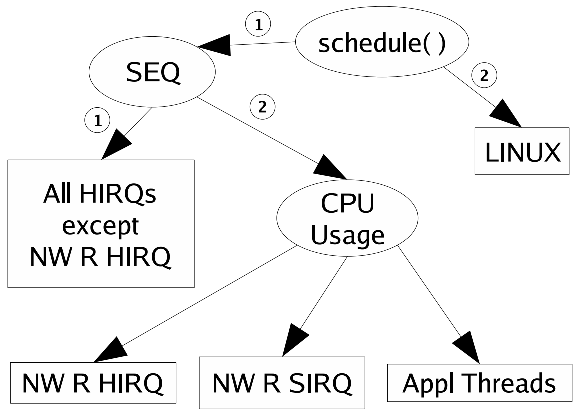

Figure 2. CPU Isolation Control SSDF

with the exception of the network receive hard IRQ thread which is under CPU share control. The order in which the SDF checks the hard IRQ handler status matches their priority relationships under default Linux scheduling, since we wish to modify system semantics as little as possible while achieving the desired goal of isolating computations from interference. The bottom of this subtree is the group controlling the network receive hard and soft IRQ handlers as well as the request server target thread, under a CPU share SDF. This SDF constrains the CPU percentage used by each thread and by the group as a whole. This SDF can be implemented various ways, but we use a simple non-preemptive and conservative accounting approach for the system described in this paper. Performance of the system under both default Linux and SDF semantics is examined in Section 5.

4. Formal Models and Verification

Our previous research [11] has shown the suitability of timed automata models for verification of timing and liveness properties in real-time system software. While that work focused on models of middleware components and their interactions, the techniques developed there are readily applicable to modeling kernel-level components and interactions. In this section we describe the timed automata models we have developed for analysis of the attack scenario that was described in Section 2.3 and illustrated in Figure 1. Section 4.1 first describes models we have developed for the system components involved in that scenario. Results of using these automata to model target system behavior under both default Linux semantics and under group scheduling control which isolates the victim thread from the effects of the request flood attack, as well as comparison of those results to measurements from the target system, are discussed in Section 5.

4.1. Computational Component Models

Figure 3 shows the model architecture we have developed for the *direct* interactions in the driving example. The ovals in Figure 3 represent system components whose semantics we represent through timed automata and the rectangles in Figure 3 represent system data structures through which packets are passed from one component to the next. In addition to the components depicted in Figure 3, we have also modeled the victim thread and scheduler components shown in in Figure 1 in Section 2.3.

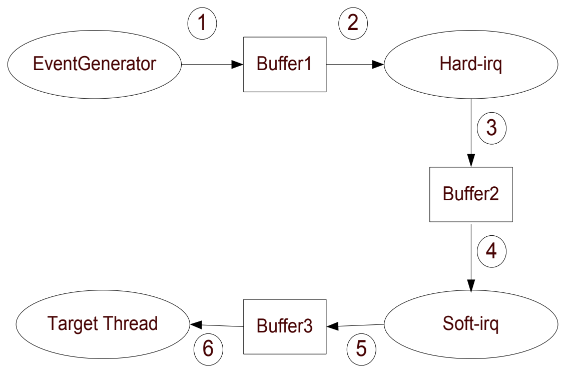

Figure 3. Packet Processing Pipeline TA

To ensure the fidelity of key temporal attributes of the system that are represented as variables in our models, we sample values for those variables from an empirically measured statistical model of several aspects of system behavior: hard IRQ inter-arrival times, hard IRQ handler execution durations, and soft IRQ handler execution durations. Techniques based in queuing theory or other detailed mathematical models also could be used to generate values for these variables in our approach, but to avoid additional validation complexity we used data measured directly from the system implementation itself. We note that this decision is another example of model/system co-design in that we obtained data from the actual system being modeled so that we could abstract away further modeling of those details.

Our models represent two different aspects of computations - their inputs and the components which execute on their behalf. Computations' inputs may include user input at a keyboard or, as in our attack scenario, network traffic arriving at an Ethernet card. While the arrival of an input requires a computational response, the arrival time is not itself under system control. We represent this behavior with a simple *EventGenerator* automaton consisting of a single state and a single transition that samples the hard IRQ interarrival time distribution to pace its transitions.

Figure 4 shows the structure of the automata for the hard IRQ, soft IRQ, and target thread components. These automata represent a mix of I/O and CPU bound actions that are coupled to the states of the buffers through which they perform I/O actions and to the scheduler automaton that governs when they are given access to the CPU.

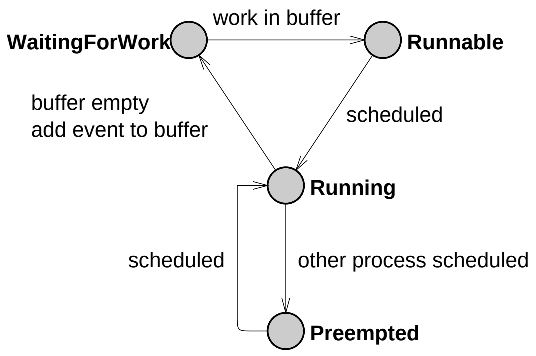

Figure 4. Target and Hard/Soft IRQ Thread TA

The CPU bound victim thread component does not perform I/O actions. Its automaton consists only of the Running and Preempted states shown in Figure 4, and is only coupled with the scheduler automaton that governs its access to the CPU.

The execution of these component models is ordered and controlled by a Scheduler automaton. Figure 5 shows the structure of the scheduler automaton that governs the access of the hard IRQ, soft IRQ, target thread, and victim thread components to the CPU. Note that it can dispatch both preemptable and non-preemptable processes, and that the transition from the *PreemptibleProcessRunning* state to the *SchedulingDecisionFunction* state is performed at quantum expiration or when a hard IRQ or soft IRQ is runnable, as well as under the condition for the transition from the *NonPreemptibleProcessRunning* that occurs when the process is waiting for work.

The automaton that models the group scheduling semantics described in Section 3.4 is similar to the one shown in Figure 5, except that a soft IRQ becoming runnable does not trigger the transition from the *PreemptibleProcessRunning* state to the *SchedulingDecisionFunction* state.

5. Empirical and Simulation Studies

In this section we describe a set of empirical and simulation studies we conducted to verify the design and validate the implementation of our group scheduling decision functions for isolation of computations from adversarial attack, which were discussed in Section 3.4. The empirical studies described in this Section were conducted on 1 GHz machines with 1GB RAM running KURT-Linux 2.6.15, connected by 100 Mbps switched Ethernet. The Linux based

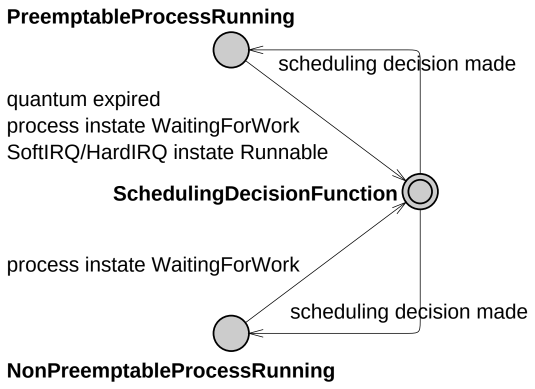
PreemptableProcessRunning
scheduling decision made
quantum expired
process instate WaitingForWork
SoftIRQ/HardIRQ instate Runnable
SchedulingDecisionFunction
process instate WaitingForWork
scheduling decision made
NonPreemptableProcessRunning

Figure 5. Default Linux Scheduler TA

experiments discussed here ran for approximately 30 seconds, with the attacking thread sending request packets as fast as possible. Figures 4 and Figure 5 in Section 4 were obtained from implementing our models in UPPAAL 3.6, but the verification studies in this section were run using exhaustive simulation of our models implemented in IF 2.0, on a 2.8 GHz processor with 2GB of RAM.

5.1. Empirical Profiling

As we noted in Section 4, to ensure the fidelity of key temporal attributes of the system our models sampled empirically collected distributions of timing values. We collected data about several aspects of request packet processing, including: (1) the distribution of Ethernet hardware interrupt inter-arrival times, (2) the distribution of hard IRQ handler execution times, (3) the distribution of soft IRQ handler execution times, (4) the distribution of the number of packets handled by a single invocation of the soft IRQ handler, and (5) the distribution of network packet handler execution times.

Analysis of these data revealed that execution time of both the network hard IRQ handler and the network packet handler routines exhibits low variance. The attacking thread transmitted over 7 million packets, which generated roughly 1.3 million network hardware interrupts. All but 1000 interrupt inter-arrival periods were less than 150 microseconds, with a fairly normal distribution between 25 and 150. The execution time of the handler was below 20 microseconds in all cases, and below 6 microseconds for all but 1800. The network driver packet handler executed over 7 million times for which all but 14 had execution intervals of less than 41 microseconds, and all but 200 thousand were below 9 microseconds. The distribution of network soft IRQ handler execution interval lengths was broader because each execution could process from one to over 20 packets.

We also collected data describing the request flood attack scenario. We collected data about the percentage of the

CPU used by the threads of interest: (1) network hard IRQ handler, (2) network soft IRQ handler, (3) target thread,, and (4) victim thread which is CPU bound. Under the attack scenario the CPU use by the first three threads is influenced by the number of request packets arriving at the victim system. The CPU use by the victim thread is influenced through the CPU scheduling interaction channel.

5.2. Studies with Default Linux Scheduling

To verify the fidelity our design and validate our implementation of the group scheduling decision functions described in Section 3.4, we examined the attack scenario in two ways: (1) through the timed automata models described in Section 4; and (2) through a simple client/server implementation. In both cases, the victim thread was a CPU bound greedy process and the target thread consumed request packets. The attacking thread was on another system generating request packets as quickly as possible.

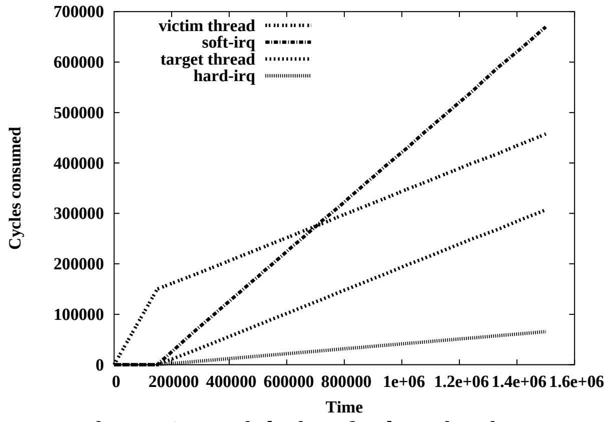

Figure 6. Modeled Default Behavior

Figure 6 shows the output from exhaustive simulations under default Linux scheduling semantics, using values drawn from the empirical distributions discussed in Section 5.1. In these simulations, as in the validation experiments we conducted with our implementation, the victim thread is allowed to run unimpeded for a while before the attack begins. Once the attack begins, our simulations predict a sharp decrease in the victim thread's CPU utilization, and a sustained rate of CPU utilization by the other components, at the victim thread's expense. Unfortunately, in the simulations presented here and in Section 5.3, we were unable to obtain output from IF for runs longer than 1.5 simulated seconds, much shorter than the 30 seconds of the actual experiments. However, this duration was sufficient to evaluate the relevant behavior.

Figure 7 shows CPU use in the actual attack experiment under default Linux scheduling semantics. The horizontal axis is time in nanoseconds, and the vertical axis is accumulated CPU use in the same units. Note that the victim thread executes briefly at the beginning of the experiment at a much higher CPU utilization than after the attack begins, as indicated by its major change in slope. The CPU utilization of the hard IRQ, soft IRQ and target threads all increase as the packet flood begins, indicating that they are interacting with the victim thread through the CPU. Under default Linux semantics, when the request flood began, the CPU utilization of the network soft IRQ thread rose from essentially zero to roughly 50%, abruptly reducing the CPU consumption of the victim, as demonstrated by its abrupt reduction in slope at the moment the attack began. The large increase in the soft IRQ thread utilization reveals that it is the channel though which the majority of interference with the victim thread is occurring, as the simulations shown in Figure 6 predicted.

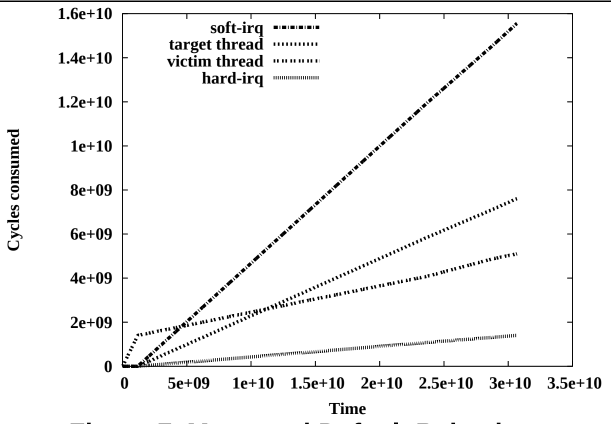

Figure 7. Measured Default Behavior

5.3. Studies with Group Scheduling

Figure 8 shows the output from exhaustive simulations under the group scheduling semantics we designed, again using values from the empirically obtained distributions discussed in Section 5.1. As in the previous studies, the victim thread ran for a while before the attack began. The simulations predict that the victim thread's CPU utilization will suffer a minor (though non-zero) decrease, indicating both the efficacy and limitations on the ability of our approach to isolate computations from adversarial attack.

Figure 9 shows the cumulative CPU use for the attack scenario under GS control which limits the CPU use of the target thread to 7%, the hard IRQ to 6%, and the soft IRQ to 12%. While the soft IRQ consumed closer to its full allocation (10.3%), the hard IRQ and target thread utilizations were noticeably lower than their specified limits (0.05% and 0.9% respectively). This result contradicts our expectation that the hard IRQ and target thread would behave greedily under GS semantics, which is how we modeled them for the

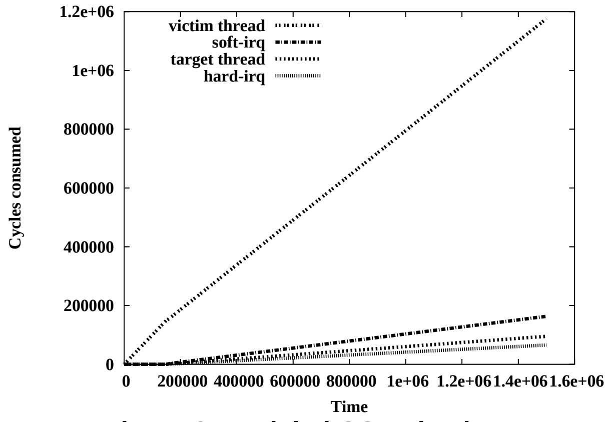

Figure 8. Modeled GS Behavior

simulations shown in Figure 8. As future work, we are investigating the network card, hard IRQ, soft IRQ, and target thread interactions in the 2.6 kernel in greater detail to explain the observed difference between the modeled and observed behavior.

Although small, the victim thread suffered a non-zero decrease in its CPU utilization when the attack began, due to the non-zero CPU allocation our GS policy made for the hard IRQ, soft IRQ and target thread. This result was predicted by our simulations shown in Figure 8.

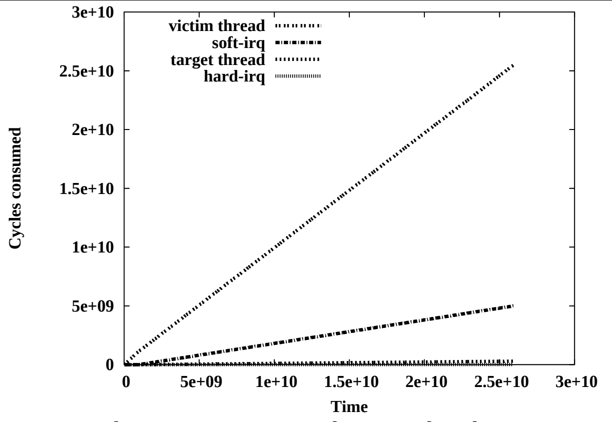

Figure 9. Measured GS Behavior

Summary of results. Table 1 shows CPU utilization by relevant threads under five cases: idle, IF simulation with default Linux semantics (DIF), execution with default Linux semantics (DL), IF simulation with group scheduling semantics (GSIF), and execution with group scheduling semantics (GSL). This table shows both the similarity and differences between the simulated and actual results, as well as the predicted and actual effectiveness of our group scheduling design. In practice, repeated refinements to the models can be used to bring the simulated and actual behaviors as close as is beneficial for particular systems.

CPU %	Idle	DIF	DL	GSIF	GSL
Victim	99	30	14.3	76	88.7
Target	< 1	20	25.1	07	0.9
SIRQ	< 1	44	51.7	12	10.3
HIRQ	< 1	04	05.1	04	0.05

Table 1. Thread Utilizations

6. Related Work

While cyber-security techniques such as network packet filtering [4, 5], data encryption [12, 13, 14], and capability management [15], help to deter malicious use of the network, inappropriate access to data, or unauthorized invocation of system functions, they do not address an attacker's ability to interfere with the execution of computations responsible for ensuring the system's correct operation.

The current state of the art in preventing interference among computations centers mainly on (1) partitioning system resources, (2) segregating computations into resource partitions, and (3) enforcing strict isolation between partitions [16]. This approach allows formal specification and verification of *separation kernels* [17] to enforce resource isolation between the partitions. While some separation kernel approaches focus solely on memory isolation, others such as the MILS kernel [18] also address isolation of process execution. However, many examples of non-adversarial interference among components of complex mission-critical systems, such as the Mars Pathfinder priority inversion problem [19], illustrate that partitioning dependencies *statically* in real-world systems is difficult.

Our approach extends earlier hierarchical scheduling frameworks [20, 21, 22] by (1) using formal models and verification techniques to determine whether properties such as isolation are achieved in each scenario; (2) taking into account the groupings of computational components involved in the various interaction channels an attacker could exploit; and (3) co-designing scheduling policies and formal models to support both tractable verification and rigorous run-time enforcement of properties such as isolation. Our automata described in Section 4.1 express refinements to the task automata used in DREAM [23], based on scenario-specific information. For example, the victim thread automaton is reduced to only two states based on its role in our driving example scenario. In earlier work, we integrated SELinux security features with group scheduling, using information from the SELinux component to help inform the decisions made by group scheduling control [24]. That research showed both group scheduling's ability to isolate competing computations' resource demands, and the utility of using information from security frameworks like

SELinux within scheduling-based approaches to isolating computations from adversarial attack.

7. Conclusions and Future Work

In this paper we have shown that integration of kernel components under group scheduling control allows isolation of computations from adversarial attack, without *a priori* partitioning of computations. Our experiences in developing this approach also have shown that careful co-design of models and scheduling policies is necessary to realize such isolation efficiently, effectively, and verifiably.

The research presented in this paper has focused exclusively on developing customized scheduling policies for isolation. An important area of future work is the extension of our earlier work on integration of SELinux security features with scheduling [24], to include similarly detailed and customized scheduling policies and verification models. With additional information about computations' expected and actual behavior as noted by the SELinux security framework, our hypothesis is that further rigor and efficiency can be realized in detecting attacks and isolating computations from adversarial or accidental interference.

References

[1] D. Bailey and E. Wright, *Practical SCADA for Industry*. Elsevier-Newnes, 2003.

[2] R. Shell and E. H. eds., *Handbook of Industrial Automation*. CRC Press, 2000.

[3] U. of Pennsylvania, "Fda/nist/nsa/nsf/nco-itrd high confidence medical device software and systems workshop." `http://www.cis.upenn.edu/hcmdss/`, jun 2005.

[4] S. Iyer, R. Kompella, and A. Shelat, "Classipi:an architecture for fast and flexible packet classification," *IEEE Network Special Issue*, vol. 15, no. 2, 2001.

[5] F. Baboescu and G. Varghese, "Fast and scalable conflict detection for packet classifiers," in *Proceedings of IEEE International Conference on Network Protocols*, 2002.

[6] 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.

[7] 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.

[8] M. Frisbie, "A unified scheduling model for precise computation control," Master's thesis, University of Kansas, June 2004.

[9] D. Mokkapati, "Evaluation of effects of a performance evaluation tool on system performance," Master's thesis, University of Kansas, april 2005.

[10] P. McKenney, "Attempted summary of "rt patch acceptance" thread, take 2." Linux Weekly News Article`http://lwn.net/Articles/143323/`, july 2005.

[11] V. Subramonian, *Timed Automata Models for Principled Composition of Middleware*. PhD thesis, Washington University in St. Louis, Technical Report WUCSE-2006-23, May 2006.

[12] P. Syverson, "Limitations on design principles for public key protocols," in *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, (Oakland, CA), pp. 62–73, may 1996.

[13] M. Winslett, N. Ching, V. Jones, and I. Slepchin, "Using digital credentials on the world-wide web," *Journal of Computer Security*, vol. 5, pp. 255–267, 1997.

[14] S. Garfinkel, *PGP: Pretty Good Privacy*. O'Reilly, 1994.

[15] B. MCCarty, *SELINUX:NSA's Open Source Security Enhanced Linux*. O'Reilly, 2005.

[16] D. Greve, M. Wilding, and W. M. Vanfleet, "A Separation Kernel Formal Security Policy," in *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2003)*, (Boulder, CO), July 2003.

[17] W. Martin, P. White, F. S. Taylor, and A. Goldberg, "Formal construction of the mathematically analyzed separation kernel," in *ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering*, (Washington, DC, USA), p. 133, IEEE Computer Society, 2000.

[18] W. Mark Vanfleet and Jahn A. Luke and R. William Beckwith and Carol Taylor and Ben Calloni and Gordon Uchenick, "MILS: Architecture for High-Assurance Embedded Computing." `http://www.stsc.hill.af.mil/crosstalk/2005/08/0508Vanfleet_etal.html`, Crosstalk: the Journal of Defense Software Engineering, August, 2005.

[19] M. Jones, "What really happened on Mars?." `www.research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html`, Dec. 1997.

[20] Regehr and Stankovic, "HLS: A Framework for Composing Soft Real-Time Schedulers," in 22^{nd} *IEEE Real-Time Systems Symposium*, (London, UK), Dec. 2001.

[21] Regehr, Reid, Webb, Parker, and Lepreau, "Evolving real-time systems using hierarchical scheduling and concurrency analysis," in 24^{th} *IEEE Real-Time Systems Symposium*, (Cancun, Mexico), Dec. 2003.

[22] Goyal, Guo, and Vin, "A Hierarchical CPU Scheduler for Multimedia Operating Systems," in 2^{nd} *Symposium on Operating Systems Design and Implementation*, USENIX, Oct. 1996.

[23] G. Madl and S. Abdelwahed, "Model-based analysis of distributed real-time embedded system composition," in *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, (New York, NY, USA), pp. 371–374, ACM Press, 2005.

[24] A. Migliaccio, T. Tidwell, C. Gill, T. Aswathanarayana, and D. Niehaus, "Group scheduling in selinux to mitigate cpu-focused denial of service attacks," Tech. Rep. WUCSE-2005-55, Department of Computer Science and Engineering, Washington University in St.Louis, 2005.