Using Smart Scheduling to Reduce the Negative Impacts of Instrumentation-based Defenses on Embedded Systems

by

Thomas Le Baron

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

April 2019

APPROVED:

Professor Robert J. Walls, Thesis Advisor

Professor Craig Shue, Thesis Reader

Professor Craig E. Wills, Head of Department

Abstract

Real-time embedded systems can be found in a large number of devices we use, including safety-critical systems. Useful for their small size and low power consumption, they are also harder to protect against state-of-the-art attacks than general purpose systems due to their lack of hardware features. Even current defenses may not be applicable since instrumentation added to defend real-time embedded systems may cause them to miss their deadlines, rending them inoperable. We show that the static properties obtained by the scheduling policies can be used as security guarantees for the tasks composing the program. By completely securing a subset of the tasks of the program only using the scheduler policy, we remove the need to add external instrumentation on these tasks, reducing the amount of extra instructions needed to entirely protect the system. With less instrumentation, the overhead added by the defenses is reduced and can therefore be applied to a larger number of systems.

Contents

List of Figures

List of Tables

1 Introduction

Real-time and embedded systems (RTES) are predominantly developed in C because it offers high performance, low-level hardware control, and is often the only language supported by the manufacturer-provided toolchain for the target device. However, C also brings a host of potential memory errors, or vulnerabilities, that are both easy for developers to make, and easy for attackers to exploit. For example, memory-corruption vulnerabilities (*e.g.*, buffer overflows) allow an attacker to overwrite certain addresses with attacker-provided values. Such vulnerabilities can be leveraged by an attacker to hijack the control flow by overwriting code pointers (*e.g.*, function pointers or return addresses). Such attacks, commonly called *control-flow hijacking* attacks, manipulate the execution of a program by redirecting control-flow transfers to either attacker-supplied code [12] or useful code sequences already in the program (*e.g.*, return-oriented programming [ROP] [15]).

One common approach to protecting against these attacks is to add instrumentation to key locations in the binary. For example, defenses that provide control-flow integrity (CFI) add instrumentation that checks if the target of an indirect jump is valid according to a control-flow graph. Similarly, stack canaries add instructions in the prologue and epilogue for each function to check for stack-based buffer overflows targeting the saved return address [8]. While instrumentation-based defenses have been explored for decades in general-purpose systems, it was not until recently that researchers began adapting these instrumentation-based defenses to real-time embedded systems.

While such instrumentation adds useful security guarantees to otherwise unprotected binaries, the instrumentation also adds overhead. In general purpose systems, the primary barrier to defense adoption is overhead with the usually accepted limit of overhead for a defense is 5% [16]; anything higher and the approach is not adopted. On the other hand, for real-time embedded systems the percentage overhead is not as important as the *schedulability* [11]. In other words, the addition of defensive instrumentation should not cause tasks to miss their deadlines.

In this work, we explore how we can make small changes to task scheduling to make large portions of defensive instrumentation unnecessary, thereby increasing the schedulability (and thus the adoption) of software defenses on real-time systems. Our approach is based on the observation that the strict scheduling semantics of a real-time system makes the order of code execution largely deterministic. For example, if we can guarantee that a task (the real-time equivalent of a thread) does not take external input and that the task cannot be preempted by another task that does, then we know that an attacker cannot manipulate the task's memory. As a consequence, we can remove all defensive instrumentation in that task. In other words, by statically ensuring that a subset of tasks cannot be preempted by others, we ensure that they cannot be corrupted in any way, removing the need to protect them with instrumentation.

We explore this idea in the context of the two most popular scheduling models for real-time systems: *fixed-priority* schedulers and *earliest deadline first* (EDF) schedulers. The contributions of this work are:

- **Safe labeling of tasks using static properties of scheduler.** We show that scheduler policies' static guarantees can be used to ensure the safety of the tasks' memory from one another, rending safe a subset of the tasks that compose the program.
- **Schedulability measure.** We propose a way to measure the schedulability of a task set, providing feedback on how far a set is from not being practical instead of a boolean test.
- **Hybrid EDF scheduler.** We show that task pushing cannot be applied directly to an earliest deadline first scheduler and propose a simple modification of the scheduler to gain the static guarantees needed to support pushing.
- **Evaluation of task pushing.** We evaluate the effectiveness of task pushing for the fixed priority scheduler in term of overhead and schedulability.

2 Background

Broadly, instrumentation-based defenses protect binaries by adding some instructions (*i.e.*, instrumentation) to check important program state at run time. For example, defenses that implement control flow integrity [2] instrument each indirect jump to check if the destination of the jump is a valid target—the beginning of a function, for instance. Similarly, defenses that provide spatial memory safety add instructions that check all pointer dereferences. While there are myriad instrumentation-based defenses that enforce a wide variety of security policies [2,6–9] they are all subject to following challenge: adding extra instructions means that code takes longer to execute and in an embedded system this might make break the real time constraints.

Below we discuss the basics of real time scheduling, the concept of schedulability, and discuss how defenses can impact real time systems. We adopt *periodic task model* to represent the execution of an embedded system [11]. This model assumes that all the tasks of the system are periodic with implicit deadlines (*i.e.*, each job must finish before the end of its period), running on a single core processor. This is the most basic embedded system model, but it applies to a wide variety of important and common devices; for example, all devices that use Amazon FreeRTOS [1], a real time operating system targeting internet-of-things devices, follow the periodic task model. More complex models allow explicit deadlines for each task or allow the program to run on multiple cores. We leave the exploration of such model for future work.

2.1 Task Systems

Each embedded system runs a program as a *task system*. Each system is composed of a finite number of *tasks*, $t_i = (e_i,p_i)$. Once the program starts, tasks are *called* one time at a specific call time and are repetitively called infinitely every period. Each task call results in a *job*. Each job of a task t_i executes for at most e_i units of time, named the *worst-case execution time* of the task and must finish its execution before its task is called again. When instrumented, the cost e_i of a task t_i is increased by a δ_i. This instrumentation cost differs from one instrumentation-based defense to another.

Each task can be in only four different states: running, ready, suspend or blocked. Only one task can run at a time and it is the role of the *scheduler* to decide which task to run among all the tasks currently in the ready state. A task is suspended when its current job finishes its execution, waiting to be called again at the end of its period. One task can block another for an arbitrary amount of time using specific system calls.

For our work, we extend the periodic task model to include the concept of a *tainted task*. Intuitively, a software-based attack requires both a software bug and input from a malicious user to trigger that bug. Therefore tasks that take external input can be corrupted and must be protected—for example, by using an instrumentation-based defense. To encode this idea, we label a task as tainted if it either 1) accepts an input directly from the user, or 2) accepts input data from a tainted task. All other tasks are labeled as *untainted* tasks. The intellectual core of this work is to understand how we can manipulate the execution order of tainted and untainted tasks to guarantee that an tainted task cannot corrupt the memory (and thus execution) of an untainted task.

How can we determine if a task is untainted or tainted? In this work, we assume that such labels are provided to us. However, we speculate that such labels could be either manually provided (not ideal) or automatically generated through static analysis.

2.2 Scheduling Policies

The scheduler is the key component in our scheme to control the execution of tainted and untainted tasks. Put simply, the scheduler picks which task to run; however, there exist enumerable policies for making that decision. In this work, we focus on the two most common scheduling policies: *fixed priority* and *earliest deadline first*.

Fixed priority scheduler. Used by the market-leading real time operating system FreeRTOS [1], a fixed priority scheduler is easy to implement and use. Each task of the task system is assigned a unique priority by the developer and the scheduler ensures that the task running is always the task in ready state with the highest priority. The priority for each task is fixed at compile time and does not change during execution.

Earliest deadline first scheduler. Given a set of tasks ready to run, the earliest deadline first scheduler will select the task with the job having the earliest deadline. In our model, the deadlines are implicit (*i.e.*, the job must finish its execution before its task is called again), therefore the earliest deadline first scheduler picks the task with the earliest task call. An earliest deadline first scheduler is therefore categorized as a *dynamic* scheduler. Consider tasks t_A and t_B that are both in the ready state at time $t = t_i$, assume that the earliest deadline first scheduler picks t_A for execution because the next period for t_A comes sooner than the next period for t_B, *i.e.*, t_A has an earlier deadline. Now imagine that at some later point in time $t = t_j$, the scheduler once again has to choose between executing t_A and t_B. It may happen that t_B has the earlier deadline now and thus the scheduler will now pick t_B over t_A. Compare to a fixed priority scheduler, there is no static ordering in the priorities of the tasks.

2.3 Schedulability

Regardless of the specific policy, a set of tasks is said to be *schedulable* if all tasks always run to completion before the end of their respective periods [5]. In other words, a schedulable task set is guaranteed to meet all real time deadlines. Schedulability is often the most important factor to consider when deploying real time systems.

Both fixed priority and earliest deadline first schedulers are also *preemptive* schedulers: they can pause the running task, save the task's execution context, run another task, and then resume the first task after the second task finishes its execution. The ability to preempt task execution is important as it allows a larger number of task sets to be schedulable—intuitively, this is because preemptive schedulers allow for a higher degree of flexibility.

2.4 Impact of Instrumentation-based Defenses on Schedulability

Most instrumentation-based software defenses are added automatically by the compiler or through a process of binary rewriting. In the context of real-time systems, this means that such defenses are not accounted for during the schedulability analysis for the task system—this is especially true for defenses based on binary rewriting since the primary allure of binary rewriting is that it can be applied without access to the source code and without cooperation by the developer.

Ideally, a task set that is schedulable without protection will also be schedulable when instrumented. However instrumentation adds overhead to each task, increasing the execution time. With each increase in execution time, tasks' jobs become more likely to not finish their execution before the task is called again. In short, a defense may render the task set unschedulable and, thus, unusable by a real time system.

The goal of this work is to decrease the chance that an instrumentation-based defense will render a task set unschedulable and, by extension, increase the adoption rate of such defenses. Our proposed techniques leverage the following key idea: we can leverage the strict execution policies of a real time scheduler to greatly reduce the amount of instrumentation needed to provide important security properties. In particular, by ensuring that an tainted task can never preempt an untainted task, we can guarantee that the former cannot manipulate the memory of the latter without the need for run time checks (*i.e.*, instrumentation). We assume that the code section of each task is not writable.

2.5 Testing for Schedulability

The primary goal of this work is to increase the number of task sets that are schedulable with instrumentation-based software defenses. To test whether a task set is schedulable, we have to apply one of two tests depending on if the set is running under a fixed priority or earliest deadline first scheduler.

The task priorities under a fixed priority scheduler play a significant role in the schedulability of a set. A previously schedulable task set may become unschedulable if the task priorities are changed. Similarly, an unschedulable set may be schedulable with new priorities. In a fixed-priority scheduler the assignment strategy is to assign priorities based on tasks' period [11]: the smallest the period, the highest the priority. This assignment is *optimal*: if a set is schedulable with a priority assignment then it must be schedulable with this optimal one; if it is not schedulable with the optimal assignment, then it is not schedulable with any other.

The schedulability of a set scheduled with a fixed priority policy can be tested by simulating the worst case execution. The worst calling situation for the jobs is to call their all tasks at the same time [11]: when all tasks are called at the same time, the jobs resulting of these calls will reach their worst end of execution time. By demonstrating that all jobs finish their execution before the end of their task's period in this particular case, we prove that no task will ever miss its deadline, and we know the set is schedulable [4]. We use this schedulability test to evaluate our proposed techniques in Section 3.2.

For the earliest deadline first scheduler, schedulability is easier to test: a task set is schedulable if its *total utilization* is no greater than 100% [11]. The *utilization* of a task i is given by $u_i = e_i/p_i$ and the total utilization of the set is the sum of the all tasks' utilization $U = \sum_{t_i \in \tau} u_i$.

3 Safe Labeling for Fixed Priority Scheduler

In this section we show how adjustment of tasks priorities in the fixed priority scheduler can be used to ensure the safety of a subset of the task set—removing the need to instrument those tasks and increasing the schedulability. We propose multiple algorithms for determining the new priority assignment and one algorithm which measures the schedulability of a task set.

3.1 Task Pushing

As discussed in the background section, a fixed priority scheduler always picks the task with the highest priority among the tasks ready to run. It is a static scheduler because the priority of a task does not change while the program executes. More concretely, given a task set of two tasks, t_A and t_B that are ready to run, if t_A has a higher priority than t_B, t_A will always be chosen. This implies that 1) t_A will never be preempted by t_B, therefore no job of t_B will ever be able to overwrite the memory used by a job of task t_A, and 2) t_A can preempt t_B, so a job of t_B can have its memory corrupted by a job of t_A. Our observation is that the priority of the task unambiguously determines which tasks can preempt the others.

In this situation where the priority of t_A is greater than the priority of t_B, if t_A is untainted (*i.e.*, it has no user inputs) then even if there are memory errors in t_A, an attacker will not be able to exploit them directly. Since t_A has a higher priority than t_B, then t_B will never be able to preempt the execution of t_A and therefore cannot be used to exploit possible bugs in t_A. Therefore t_A's memory cannot be corrupted, there is no need to instrument it and only the t_B needs to be protected. On the other hand, if t_B has no user inputs but t_A does, t_B still needs to be protected since t_A can be used by an attacker to overwrite the memory of t_B and therefore exploit possible memory errors present in the task. In this case, we propose to increase the priority of t_B to be greater than t_A. We call *task pushing* the action of increasing the priority of an untainted task in order to secure its memory. We increase the priority of untainted tasks so that they become greater than the priority of each tainted task of the set. When pushed, the untainted tasks is referred to as *safe*: its memory is totally safe because of the scheduling policy and no extra instrumentation is needed. Initially, the priority is only used to

make the set schedulable. Here we show that priorities can also be used to statically guarantee the safety of the memory of a subset of the task set executed.

Task pushing impacts schedulability in two opposite ways. On one hand, it may decrease the likelihood of schedulability by changing the priority assignment to one that is non-optimal. On the other hand, it may improve the schedulability by removing the need for instrumentation on the pushed tasks. Since the execution time of each safe task is reduced compared to their instrumented version, the set is more likely to be schedulable.

3.2 Pushing Algorithms without Schedulability Measurement

Since pushing tasks change the optimal priority assignment, there may be cases where pushing tasks makes a schedulable set unschedulable. Pushing all untainted tasks is not necessarily the best decision. Therefore, we need to choose which tasks to push.

Each algorithm below takes two inputs: the initial task set, without any protection, ordered in decreasing priority when following the optimal priority assignment, and the set of untainted tasks, ordered the same way. As an output, the algorithm returns a subset of the untainted tasks set which correspond to the tasks to be pushed. The next step is to assign the priority in a decreasing order, picking first the task present in the output of the algorithm and then the ones not in this output set. After that, we instrument only the tasks not present in the output set of the algorithm, resulting in a fully secured task set with reduced instrumentation.

Tasks can be safe if their priority is higher than the one of the tainted tasks, but the ordering between the unsafe tasks does not matter for their security. Therefore we choose to push the tasks so that their final ordering follows the optimal priority assignment.

3.2.1 Labeling Without Pushing

Some untainted tasks may have a higher priority than all tainted ones even under the optimal priority assignment. In this situation, we can avoid instrumenting them without modifying the order of priority of the task. Since we keep the optimal order, there are no drawbacks on doing so. Therefore these tasks will always be picked to be pushed, even by the other pushing algorithms.

The following pseudocode describes how we select these tasks.

Algorithm 1: Freewin

```
   input  : initial task set, set of untainted tasks
   output : set of tasks that do not need instrumentation
1  foreach task t_i in initialSet do
2      if t_i is untainted then
3          add t_i to outputSet;
4      else
5          break;
6      end
7  end
8  return outputSet
```

The untainted tasks that are safe without pushing will always be in the set of safe tasks since they will never reduce the schedulability of the set. Therefore, labeling such tasks as safe is the first step of all the following pushing algorithm we present. Only labeling tasks as safe without pushing removes the risk of making the set unschedulable but reduces the effectiveness of the labeling operation. The following algorithms present different strategies to decide how to pick the untainted tasks to push.

3.2.2 Binary Period Pushing

We observe that pushing a task with a low initial priority is more likely to have a higher negative impact on schedulability than pushing a task with a high initial priority. As we saw in the previous algorithm, pushing tasks with the highest priorities has no negative effect on schedulability. On the other hand, if the only untainted task of a set is the one with the initial smallest priority, then pushing it will delay all the other tasks of the set, making these tasks more likely to have a job missing its deadline. This observation suggests that a simple greedy algorithm might work well. In order of initial priorities, the algorithm pushes as many tasks as possible and stops when pushing one more will break the schedulability of the set.

Algorithm 2: Binary period

```
   input  : initial task set, set of untainted tasks
   output : set of tasks to push
 1 outputSet = Freewin(initialSet);
 2 wasSchedulable = isSchedulable(safeSet(inputSet, outputSet));
 3 foreach task t_i is untainted and not in outputSet do
 4 |   testSet = safeSet(inputSet, outputSet + t_i);
 5 |   if testSet is schedulable or wasSchedulable = False then
 6 |   |   add t_i to outputSet;
 7 |   else
 8 |   |   break;
 9 |   end
10 |   if testSet is schedulable then
11 |   |   wasSchedulable = True
12 |   end
13 end
14 return outputSet;
```

The $safeSet(set1, set2)$ function is used to create an labeled version of the $set1$ in which all the pushed tasks are the tasks from $set2$.

3.3 Schedulability Measurement

For the previous algorithm, we assume that the initial order of tasks (sorted by period) is the same as if the task were sorted by impact on schedulability, from the best impact to the worst. This is obviously not true since the impact on schedulability also depends on the overhead added on each task by the instrumentation: a task with no instrumentation will never bring the set closer to schedulability if pushed since doing so will not reduce its execution time. There are situations where it may be better to push a task with a large period, even if it means delaying a lot of other tasks. Let a task set be (t_1, t_2, t_3) with $t_a = (e_a, p_a, \delta_a)$ with $p_1 \leq p_2 \leq p_3$, e_i being the execution needed including instrumentation cost and δ_i being the cost of the instrumentation of the task t_i. If t_2 is pushed, t_1 will be delayed by $e_2 - \delta_2$. If t_3 is pushed, t_1 and t_2 will be delayed by $e_3 - \delta_3$. Since the period of each task is independent from its execution time and the cost of their instrumentation, we may have $e_3 - \delta_3 < e_2 - \delta_2$. It may happen that pushing t_2 makes a job of t_1 miss its deadline while pushing t_3 does not.

While a boolean test on schedulability is useful, what we really need is a test that tells how much closer an unschedulable set is to schedulability after pushing that task. In other words, if we can measure the magnitude of the impact, rather than limiting ourselves to a simple boolean response, we can develop better selection algorithms.

We simulate the worst calling case for a set under the fixed priority policy, when all tasks are called at the same time, and define the *schedulability of a task*. In a schedulable set, a task's schedulability measurement

corresponds to the amount of time between the end of its period and the end of its job's execution in its worst execution situation. We can then define the schedulability of a schedulable set as the minimum of the schedulability measurements of its tasks. For a schedulable set, this measurement will always be positive.

An unschedulable set is composed of a schedulable subset of high priority tasks (which may be empty), of one task that cannot finish its first execution before the end of its period (in its case, the difference between its period and its worst execution time is negative), possibly followed by other tasks (which can be totally starved). The schedulability measurement of an unschedulable set is composed of the index of the unschedulable task and its schedulability measurement.

Algorithm 3: Schedulability measure

input : task set
output : pair (blocking task, schedulability of blocking task)

```
1 foreach task t_i in input set do
2     taskSchedulability = task.period − worstExecutionTime(task, inputset);
3     if taskSchedulability ≥ 0 then
4         append {'blockingTask' : t_i, 'schedulability' : taskSchedulability} to positiveMeasuresList;
5     else
6         return {'blockingTask' : t_i, 'schedulability' : taskSchedulability};
7     end
8 end
9 return the entry of positiveMeasuresList with the smallest schedulability;
```

If the set is schedulable, the *schedulability* measurement returned will be positive. To compare the schedulability under two different priority assignments of the same set, both assignment making the set schedulable, we just compare this measurement: the smallest is assigned to the set the closest to be unschedulable. To compare the schedulability of two unschedulable sets composed of the same tasks with different priority assignments, we need first compare the index of the blocking task before comparing their schedulability. The set closest to being schedulable is the one with the highest blocking task index. If these indexes are equal, we compare the schedulability score of these tasks to keep the set with the measurement the closest to be positive.

3.4 Pushing Algorithms with Schedulability Measurnte

The schedulability measurement gives more information than a function that returns a boolean on the schedulability of a set used in the previous algorithm. It tells how far a set is from the schedulability limit. With this, we can define other algorithms based on the schedulability measurement induced by pushing the untainted tasks. The first algorithm still make the greedy decision of pushing a task in order of initial priority while the second makes it in term of schedulability impact (best impact first).

3.4.1 Schedulability Period Pushing

As in Section 3.2.2, the tasks are sorted by initial priority but are only pushed if doing so increases the schedulability of the set (schedulability measurement greater than former best) or if it keeps the set schedulable (schedulability measurement positive). Since this algorithm can spot and avoid pushing the tasks which can drive the set further from getting schedulable, it will always give a better result than the previous pushing

algorithm.

Algorithm 4: Schedulability period

```
   input  : initial task set, set of untainted tasks
   output : set of tasks to push
 1 outputSet = Freewin(initialSet);
 2 currentBestMeasure = schedulabilityMeasure(safeSet(inputSet, outputSet));
 3 foreach task t_i, untainted and not in outputSet do
 4 |   testSet = safeSet(inputSet, outputSet + t_i);
 5 |   testMeasurement = schedulabilityMeasure(testSet);
 6 |   if testMeasurement > 0 or testMeasurement better than currentBestMeasure then
 7 |   |   add t_i to outputSet;
 8 |   |   currentBestMeasure = testMeasurement;
 9 |   else
10 |   |   break;
11 |   end
12 end
13 return outputSet;
```

3.4.2 Pure Schedulability Pushing

With the schedulability measurement, it is possible to classify an untainted task by impact on schedulability and push them in order of decreasing negative effect. We then have an efficient algorithm we can use to increase the schedulability of the sets. We measure the schedulability resulting from pushing each untainted task at a time. Only the best task is recorded as pushed. This continues until there is no untainted task left to push, or if pushing any one reduce the schedulability of the set or make it unschedulable. Since we do not sort the untainted tasks by periods, it is possible for this algorithm to return a set less schedulable than the

previous algorithms.

Algorithm 5: Pure schedulability

```
input  : initial task set, set of untainted tasks
output : set of tasks to push
outputSet = Freewin(initialSet);
while an untainted task is not pushed do
    currentBestMeasure = schedulabilityMeasure(inputSet, outputSet);
    currentBestTask = NULL;
    foreach task t_i, untainted and not in outputSet do
        testSet = safeSet(inputSet, outputSet + t_i);
        testMeasurement = schedulabilityMeasure(testSet);
        if testMeasurement > 0 or testMeasurement better than currentBestMeasure then
            currentBestTask = t_i;
            currentBestMeasure = testMeasurement;
        end
    end
    if currentBestTask is not NULL then
        add currentBestTask to outputSet;
    else
        break;
    end
end
return outputSet;
```

All these algorithms focus on improving the schedulability of the sets. If a set is already schedulable while being fully instrumented, improving its schedulability measurement is not needed. Instead, we would need to maximize the overhead reduction to speed up the system as much as possible. We need a way to measure the overhead of a task set. With this measurement, we will be able to compare the different overhead reduction these pushing algorithms can reach.

3.5 Overhead Measurement

In our task models, each period p_i is measured as a certain amount of time unit. Since this measurement is an integer for all tasks of a set then there is a defined period P for the entire set. The *hyperperiod* of a task set is defined as the product of all the periods of tasks of the set $H = \sum_{t_i \in \tau} p_i$. If all tasks are called at the same time at $t = 0$, then they will also be called at the same time again at $t = H$ since all tasks will have finished a integer number of periods.

The measure of the execution time is as follows: after we compute the hyperperiod of the tasks of a set by multiplying each period, we measure the total execution time of each task during this cycle, taking into account that tasks need to be executed multiple times. Since we only measure the overhead for a schedulable set, we are assured that the measurement we are looking for exists and is lower than the hyperperiod of the set. This execution time measurement is done twice per pushing algorithm: once without instrumentation and once with the unsafe tasks instrumented. The overhead induced by the instrumentation is simply the difference between the two measurements.

3.6 Brute-force Algorithm

It would be possible to use an algorithm similar to the previous one (*Pure schedulability*, Section 3.4.2) to efficiently pushs the tasks in a large number of sets. In practice, labeling as safe some tasks of a program will occur only once. According to the sporadic task model we use [5], the size of task sets are rarely composed of more than forty tasks, and therefore, we consider a brute-force priority assignment to be practical.

The brute-force algorithm must first find from all set of task pushed possible, which one of them make or keep the task set schedulable. Among them, if at least one exists, the algorithm must return the set with the least overhead.

Algorithm 6: Bruteforce

input : initial task set, set of untainted tasks
output : set of tasks to push

```
1  removedOverhead = 0;
2  initialize outputSet;
3  for all possible subset of the set of untainted tasks do
4  |   testSet = safeSet(inputSet, subset);
5  |   if testSet is schedulable and overheadMeasurement(inputSet, testSet) > removedOverhead then
6  |   |   outputSet = subset;
7  |   |   removedOverhead = overheadMeasurement(inputSet, testSet);
8  |   end
9  end
10 return outputSet;
```

4 Safe Labeling for Earliest Deadline First Scheduler

The fixed priority scheduler is easy to implement and has a low scheduling overhead. As we saw, this policy can be used to ensure statically that some tasks will never be preempted by other, allowing us to label some tasks as safe. Nevertheless, these static guarantees come at a price: even following the optimal priority assignment, tasks sets may be unschedulable with this scheduler while they may be schedulable with another one. On the other hand, earliest deadline first is an optimal scheduler: if a set is not schedulable with this policy, it cannot be schedulable with any other one [11]. Unlike fixed priority, earliest deadline first is a *dynamic* scheduling policy: the choice of which task runs between two tasks t_A and t_B depends on runtime characteristics instead of a static value like the priority. In the case of earliest deadline first, the scheduler chooses to run the task that has the job with the closest deadline: which task that is called again first.

Due to this flexibility, tasks sets are more likely to be schedulable on the earliest deadline first scheduler than on the fixed priority scheduler. It also removes the possibility of pushing tasks because, with the earliest deadline first policy, there is no fixed ordering in tasks: each of them can have the priority on any other during the execution of the program. Only the period of the tasks can give an ordering on preemption: between two tasks t_A and t_B, only the tasks with the shortest period will be able to preempt the other. To preempt t_B, t_A needs to have the priority, therefore it needs to have its period finishing before t_B, but also need to let t_B start its execution. This second requirement asks for t_A to be called after t_B is called, which implies that the period of t_A must be shorter that t_B's period. Nevertheless, the period of a task is fixed by the developper and we cannot use it for safe labeling.

To merge the best of the two schedulers (high schedulability and static properties) we propose a new scheduling policy called *Hybrid EDF*.

In this Hybrid EDF scheduler, we label a subset of tasks as *critical* and the rest as *non-critical*. The scheduler will always prioritize running critical tasks before the non-critical tasks. If only critical tasks or

only non-critical tasks are waiting to run, the scheduler acts like a simple earliest deadline first scheduler: it runs the task with the earliest deadline. If both critical tasks and non-critical tasks are waiting to run, the scheduler always picks a critical task to execute without comparing its deadline with the deadlines of the non-critical tasks. With this policy, untainted tasks can be labeled as safe if they are in the critical set while all tainted tasks must be in the non-critical set. Other types of hybrid schedulers have been studied [10], [11], but to the best of our knowledge, none can be used to label an arbitrary set of tasks as critical.

4.1 Testing the Schedulability of Hybrid EDF

To evaluate the Hybrid EDF policy we need a way to test the schedulability of a task set. Neither the schedulability test for fixed priority schedulers nor the one for simple earliest deadline first one can be used in this situation. On fixed priority schedulers, the worst calling convention is when all tasks are called at the same time [11] so we only need to check the schedulability of each task on its first period only. For the hybrid scheduler, calling all tasks at the same time is not necessarily the worst case, so ensuring that the first job of each task meets its deadline is not sufficient to prove the schedulability of the set. A set is schedulable on an earliest deadline first scheduler if its total utilization (defined in Section 2) is not greater than 100% [11]. For our hybrid EDF scheduler, task sets with less than 100% of total utilization might still be unschedulable.

Since we cannot use the same test to determine directly if a task set is schedulable, we try to simulate it. Simulating the execution of a task set to determine if it is schedulable must be done for the worst possible case: we call the tasks so that one of their job finishes as late as possible and verify if this job meets its deadline. The set is schedulable if it is the case for all the tasks. We therefore need to determine what is the worst case for the hybrid scheduling policy.

Lemma 1: *under the hybrid scheduler, a critical task missing its deadline necessary means that all non-critical tasks miss their deadlines too.*

Proof: in task set $(t_{c1},...t_{cn},t_{d1},...,t_{dm})$, t_{ci} is critical and t_{di} is not. Assume that there exists a j such that t_{cj} misses its deadline. Since the tasks t_{c1}, ..., t_{cn} are scheduled between them with an earliest deadline first scheduler, it means that the total utilization of the subset (t_{c1}, ..., t_{cn}) is greater than 100%. The total utilization of the entire task set will therefore always be greater than 100% if one critical task misses its deadline, and all the non-critical tasks will miss their deadlines as well. □

With this lemma, we ensure that the task set being unschedulable implies that the non-critical tasks miss their deadlines. It also means that if all non-critical tasks meet their deadlines, the whole set is schedulable. We can therefore only check the schedulability of the non-critical tasks. In the case where there are only critical tasks, there is no need to go further: with only critical tasks the scheduler acts like a pure earliest deadline first scheduler, for which we already have a schedulability test. We now assume that there is at least one non-critical task in the task set.

A non-critical task can be delayed either by another non-critical task or by a critical task. We start by looking for a simulation in the case where there are no critical tasks. Between the non-critical tasks, the scheduler acts as a pure earliest deadline first scheduler, but we still need a schedulability test by simulation. There are earliest deadline first schedulability tests [3] that do not compare the total utilization with 100%, designed to relax some assumptions of our task model, but do not simulate the execution of the set. They only work for a pure earliest deadline first scheduler and cannot be used further to test the schedulability on the hybrid scheduler.

4.2 Worst case of earliest deadline first scheduled task sets

Simulating tasks scheduled with the fixed priority scheduler is convenient. It is proven that the worst case for a task is for that task to be called at the same time with all other tasks [11]. In this situation, a higher priority

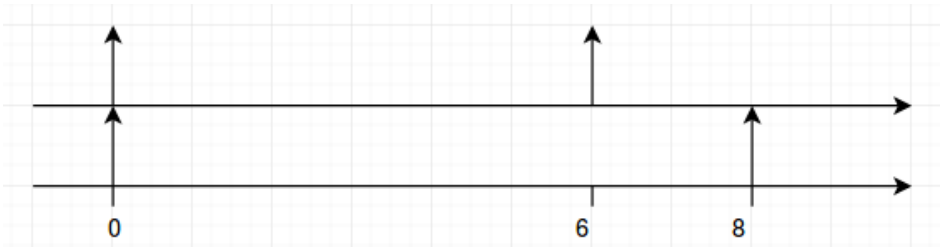

Figure 1: Example of simulation of a set of two tasks with respective periods $p_1 = 6$ and $p_2 = 8$

task will execute as many times as possible during the period of the tested task and thus it will delay the most the tested task, which will reach its worst case in terms of execution time. This is not the worst case with an earliest deadline first scheduler for a simple reason: calling all tasks at the same time may give the highest priority to the tested task at some point. Figure 1 shows as example: the horizontal axes are the time axes for each task, t_1 is the task represented on the top, t_2 is represented on the bottom. The vertical axes show when each task is called. Here, t_2 has the priority between $t = 6$ and $t = 8$. The worst way to call the tasks would reduce this time window to the minimum.

For the earliest deadline first policy, a task has the priority if its deadline is the earliest. The time window where a task has priority over any other task is between the call for its next iteration and the latest call of any other task (see Figure 1). We want to reduce this time to a minimum to simulate the worst execution situation. If all other tasks are called so that the end of their period is the exact same as the end of the period of the tested task, then this time window will be reduced to zero. When all deadlines are the same, the task to pick by the scheduler is not defined. If we test the schedulability of the tasks one by one, we have to force the script to run all other tasks before the tested task to simulate its worst case. Fortunately, testing tasks one by one is not necessary since the worst case is when all tasks finish at the same time, thus the order the tasks are picked to run does not matter.

Lemma 2: *under an earliest deadline first scheduler, if multiple task have the same deadline and the task picked to run last misses its deadline, then one task will miss its deadline no matter the order of execution used to deal with the tie.*

Proof: let a task set $(t_{c1}, ...t_{cn})$ and we call the tasks so that all of them are called again at the same time $t = T$ and t_{cj} the task with the shortest period, p_{cj}. Between $t = T - p_{cj}$ and $t = T$, the order of selection of the tasks is not defined. We note r_{ci} the remaining execution time needed by the job of the task t_{ci} to finish its execution at the time $t = T\text{-}p_{cj}$. If the sum of the remaining times r_{ci} is not greater than p_{cj},then each task will finish their execution before $t = T$, whatever the order of execution is. On the contrary if it is not the case, then at least the last picked task's job will miss its deadline. □

It is therefore not needed to test the schedulability of each task separately. With an arbitrary pick order in the case of a tie, if each job manages to finish its execution before their task are all called again, then all tasks are schedulable and the task set is schedulable.

There is a significant difference between this test and a feasibility test for an fixed priority scheduler. For a fixed priority scheduler, each task is *called* at the same time, and we check that each first job meets its first deadline. For the earliest deadline first scheduler, the worst case is when all tasks *finish* one of their period at the same time, so we need to determine how to call them so that this situation is achieved. If we call all tasks at the same time at $t = 0$ then we know that they will all be called again at $t = H$ with H the hyperperiod of the set. To verify that all the task meet their deadline at $t = H$ we have to simulate the execution of the set during the whole window $[0, H]$.

This solution is not practical at all. A simple example of task set, generated using the same task generation system than the Section 5, is composed of only three tasks with as periods 3,960,000, 5,280,000 and 7,260,000

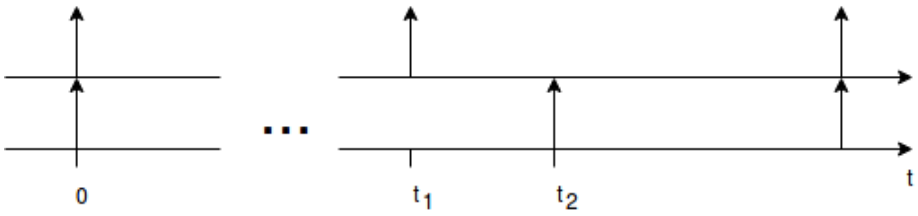

Figure 2: Example of existence of the extra time window for the second task between $t = t_1$ and $t = t_2$

cycles. The hyperperiod lasts for around 10^{20} CPU cycles, which correspond to the simulation of almost 10^{14} task calls to process to test the schedulability of only one set. Not only is the simulation of a set of size 3 already not practical, but some task set generated can have more than 30 different tasks.

4.3 Under-approximation implementation

To reduce the testing time we want to come up with an approximate test. If we only simulate a lower number of periods, instead of the whole hyperperiod, we will have a less accurate result, but we will get the result in a shorter amount of time. We need this test to be an under-approximation: we must simulate a case worse than what it is so that a unschedulable set will never be mistaken for a schedulable one. As in the previous test, we need all the tasks to finish at the same time to get the worst case. Unlike the previous test, we reduce the time window of study from $[0, H]$ to $[H - p_n, H]$, where p_n is the longest period of the task set. For example using the task set of Figure 2, the under-approximation will only start the simulation the execution of the two tasks at $t = t_1$.

Each full period of each task is easy to generate with the information we have now: we can exactly simulate them without approximation. The approximation part comes from the extra time needed for each task to complete the whole test window. Since this window does not necessarily correspond to an integer number of periods for each task, some of them have an extra time window we need to deal with. For example in Figure 2, the second task has the priority over the first one between $t = t_1$ and $t = t_2$, but we do not know if the job of the task t_2 released at $t = t_2 - p_2$ (with p_2 the period of the second task) still need to run after $t = t_1$. The approximation comes from the fact that we do not have enough information to know for each task what is the amount of execution left for the instance of this task called just before the beginning of the test window. We only know that the maximum execution time in each of the extra time window for each task is at most equal to the execution of one of their job, *i.e.* is at most equal to their execution cost.

We start by computing the length of the extra time window for each task t_i in which the execution is approximate:

$$extra(task_i) = p_n - \left\lfloor \frac{p_n}{p_i} \right\rfloor * p_i$$

Here, p_n is the longest period of the set, which corresponds to the length of the test window.

We place ourselves in the worst case, *e.g.* we assume that the extra time of each task is used as much as possible. To simulate this, we sort the tasks by increasing extra time (see Figure 3). For each task, we compare the execution time and the amount of time available in this extra time window, taking in account that previous tasks will run during this window: the minimum of these two values is the time used by the current task in its extra time window.

$$extraExec_i = min(e_i, extra(task_i) - \sum_{j<i} extraExec_j - \sum_{j<i} fullExec_j)$$

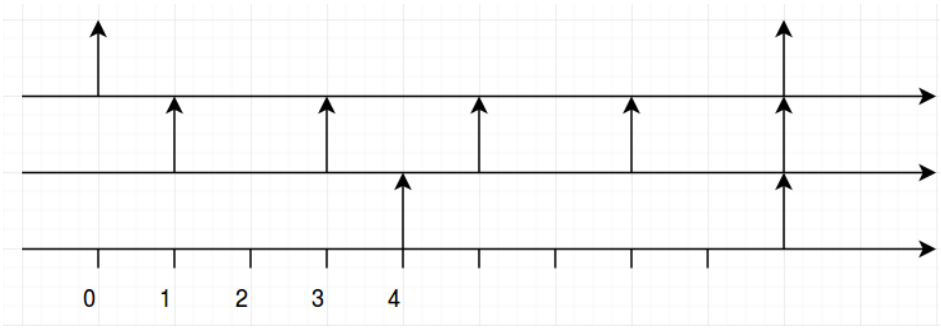

Figure 3: Example of task calling convention for under-approximate schedulability test

For the first task, $extraExec_1$ is equal to 0. Then for each task, the available execution time for in its extra time window is equal to the length of this time window ($extra(task_i)$), from which we remove the extra execution ($extraExec_j$) of the previous tasks and their potential full executions ($fullExec_j$). It is indeed possible for a task with a short period to have a very small extra time window. In this case, the tasks with longer periods may have to let the shorter task run multiple full executions during their own extra time window. For instance in Figure 3, the available extra time of the third task is reduced by the extra execution of the second task but also by one full execution of the second task : $extraExec_3 = min(e_3, 4 - extraExec_2 - 1 * e_2)$.

5 Evaluation

In this section we evaluate the effectiveness of task pushing on the fixed priority scheduler and the accuracy of the under-approximate test of scehdulability of the earliest deadline first scheduler.

5.1 Evaluation of Task Pushing on Fixed Priority Scheduler

The point of task pushing is to label tasks as safe so that there is no need to protect them by instrumenting them. In this section, we answer the following questions: how much instrumentation can task pushing save? how many times saving this much overhead can make protected set schedulable?

We need to pick a defense for embedded systems based on code instrumentation. RECFISH [17] is a CFI based defense for embedded systems. Since the destination of an indirect jump is written in memory dynamically, it may be overwritten by an attacker and be used to control the flow of execution. RECFISH instruments each indirect jumps to ensure at runtime the integrity of their target.

We run the evaluation using the standard sporadic tasks model also used in the RECFISH evaluation, which is a common study in work on real-time systems [11]. The standard sporadic task model distinguishes the task sets by classifying them by their values in different characteristics: the total system utilization in $U \in \{0.05, 0.1 \ldots, 1.0\}$, the per-task utilization is picked uniformly in a range classified as *light* ($[0.001, 0.1]$), *medium* ($[0.1, 0.4]$) or *heavy*($[0.5, 0.9]$), the periods of all tasks are chosen uniformly from either $[3, 33]$ ms (*short*), $[10, 100]$ ms (*moderate*), or $[50, 250]$ ms (*long*), the amount of overhead for indirect jump instrumentation is picked $\in \{33, 50\}$ cycles, the number of indirect branches can be *None* or are uniformly chosen at a rate of one indirect branch among $[10^3, 10^5]$ cycles (*common*), $[10^6, 10^7]$ cycles (*rare*), or *bimodally* between the two distributions none (90%) and common (10%). Similarly, the total number of functions per task is chosen uniformly among $[1, 100]$ (*few*), or uniformly at a rate of one function among $[10^2, 10^3]$ cycles (*frequent*), $[10^3, 10^4]$ cycles (*moderate*), or bimodally between the two distributions moderate (90%) and few (10%).

This results in 5,760 different classes of possible task sets (we call these classes *design points*). We randomly generate 100 task sets for each design point and label each task as untainted with a probability

Algorithm	Schedulability probability
Full	84.8%
Freewin	88.2%
Binary period	88.9%
Schedulability period	90.4%
Pure schedulability	90.4%
Brute-force	90.5%

Table 1: Comparison of average schedulability after task pushing with different algorithms

of 50%. From the initial task set, without instrumentation, and the list of the taiting labels of this set, we compute the protected task set resulting from the algorithms presented in the sections 3.2 to 3.6. We test if the set is schedulable and if so measure the overhead. We can then compare the results in schedulability and overhead of a fully RECFISH-protected set and the same set after task pushing.

The RECFISH evaluation generates 1,000 task sets per design point. We had to reduce this number because the computation time for 5 millions tasks labeled by all the algorithms described in the previous section was not practical. We had to reduce the number of task sets generated to 10 per tasks points for the evaluation of the impact of taint probability (Subsection 5.2) and overhead distribution (Subsection 5.3) with the brute-force algorithm for the same reason. This is only needed for the evaluation step: even the brute-force pushing is practical on one task set (a few seconds on a standard laptop).

5.1.1 Schedulability

The schedulability results are summarized in Table 1. The first algorithm named *Full* correspond to the case where the system is protected by the default RECFISH instrumentation. There is no task pushing at all, every task is instrumented.

We can see that with a probability of task tainting of 50%, even the free win situation of Section 3.2.1 can save enough execution time to increase the chances for a task set to be protected and schedulable. This decision to not instrument untainted task that are safe can sometimes be very interesting and present no drawbacks.

Pushing tasks only using a binary schedulability test (Section 3.2.2) does not improve a lot the schedulability compare to this free win situation. The schedulability measure we proposed is a powerful tool to improve the schedulability of protected task sets as much as possible compare to the boolean schedulability test [4].

Also, the algorithm focusing on pure schedulability (Section 3.4.2) has on average the same result as the algorithm pushing tasks in their original order using the schedulability measure (Section 3.4.1). Since these two algorithms do not push the same tasks, there are some situations where one will be better than the other and others where it is not the case. More precise results still show that on average the pure schedulability algorithm is still better than by a few tasks sets out of the 576,000 generated. We notice that both the schedulability algorithms gives a probability of schedulability almost optimal by comparing them to the brute-forces algorithm. Therefore, they can be a good alternative to a brute-force approach if one needs to push tasks in a large number of task sets.

Overall, these results show that task pushing can be used to significantly increase the probability for a task set to be schedulable after instrumentation. But even if the set is schedulable with a default RECFISH protection, we may want to still push tasks, since it will reduce the overhead of the protection.

Algorithm	Overhead	Overhead (schedulable with *Full* only)
Full	15.9%	15.9%
Freewin	9.9%	10.0%
Binary period	8.6%	8.6%
Schedulability period	8.2%	8.0%
Pureschedulability	9.7%	9.3%
Brute-force	8.2%	8.0%

Table 2: Comparison of average overhead after task pushing with different algorithms

5.1.2 Overhead

The reduction of overhead by the pushing algorithms is presented in Table 2. This table compare the results of the average of the measured overhead for different algorithms. The second column is the average computed only for the sets which are schedulable by default, without task pushing. For this column, each average is calculated in the exact same collection of task sets, while the sets of schedulable task sets differ from one algorithm to another.

We saw in the schedulability evaluation (Section 5.1.1) that the schedulability period pushing (Section 3.4.1) and pure schedulability pushing algorithms (Section 3.4.2) have both a schedulability result very close to the optimal one obtained by exhaustive search. In term of overhead, we can see that there is a significant difference between the two. This is due to the fact that the tasks pushed by the first one will tend to have shorter period. Even if they have on average the same amount of overhead as any other tasks, it is better to push them since they will tend to repeat more often in the hyperperiod, reducing the overhead more than tasks with longer periods. Therefore, sorting tasks by initial priority reduces the overhead more than picking the tasks in term of the schedulability measure only.

Moreover, we notice that the schedulability period pushing algorithm (Section 3.4.1) has a resulting overhead very close to the lower bound represented by the brute-force algorithm. Since it is also the case for the schedulability, then it can be used as a great alternative to a brute-force approach to label a large number of task sets.

With a probability of tasks being untainted of 50%, the overhead can be reduced by half on average. This result, far from being negligible, shows that task pushing can be a powerful tool to reduce the impact of defenses on FreeRTOS installations and similar systems. While we may have overestimated here the probability for the tasks to be untainted, we still showed that if there are enough of them, the overhead of RECFISH can potentially be greatly improved.

5.2 Impact of the Tainting Probability

The previous evaluation has been done by assuming that the probability for a task to be untainted is 50%. Since to the best of our knowledge, there is no system to model the tasks' tainting in real time system as we defined it, we had to pick a value to perform our evaluations. But the impact of this parameter is important to determine the effectiveness of task pushing: if no tasks are untainted, then all the instrumentation must be kept, while if all the tasks are tainted, no instrumentation at all will be needed for the system to be safe. We want to determine the effectiveness of task pushing for different probabilities of task tainting.

We measured the overhead with the brute-force algorithm for different probabilities of tainting. The results in Table 3 provide an upper bound of schedulability and a lower bound on the overhead of a protected systems for different probabilities of tainting. The fact that we had to reduce the number of tested task sets per design point to 10 to conduct our evaluation explains the small inconsistency it has with the previous results.

Algorithm	Schedulability probability	Overhead	Overhead (schedulable with *Full* only)
P = 0.25	87.8%	11.7%	11.9%
P = 0.5	90.5%	8.0%	8.4 %
P = 0.75	94.5%	4.0%	3.9 %

Table 3: Average schedulability and overhead result comparison for different probability of untainted tasks

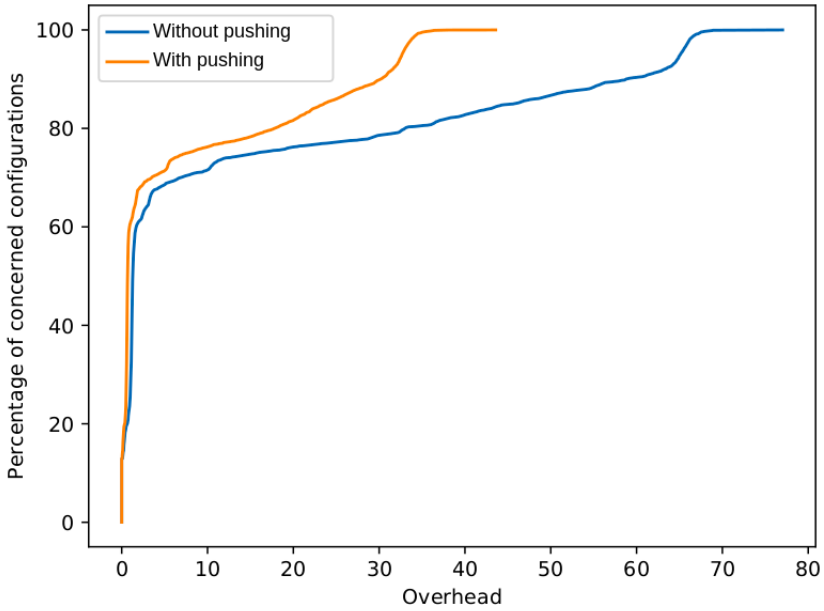

Figure 4: Comparison of the average overhead distribution of RECFISH with and without task pushing.

On average the reduction of overhead is close to linear with the probability for a task to be untainted. This can be verified by computing the average percentage of pushed tasks, which reached 95% with the brute-force algorithm. This percentage is only computed when it exists (*e.g.*, all situations where there are no untainted tasks are not counted).

5.3 Overhead Distribution

The results presented above are all averages on all the different task sets generated. As stated in the evaluation of RECFISH [17], the impact of the instrumentation differs greatly from one design point to another. Therefore we measure the percentage of schedulability and average overhead not for all the task sets generated but for each design point.

Figure 4 presents the comparison of the overhead distribution with and without task pushing by the brute-force algorithm. We can see that the RECFISH impact varies greatly from one design point to another one. A majority, around 68%, of the modeled design points have an overhead of less than 5%. On the other hand, protecting certain types of task sets slows them down a lot more, increasing by more than 60% their execution time. This other extreme situation only concerns 10% of the design points.

To explain this difference in overhead, we searched for the characteristics which impact the measure the most. We found that the design points with the least number of functions per task have the lowest overheads. We also found that task sets with high per-task utilization tend to have low overhead—all task sets with a per task utilization greater than 0.9 have an average overhead below 3%. The explanation is straightforward: the higher the per-task utilization, the lower the number of tasks per set and the lower the number of indirect

branches in the system. In other words, RECFISH overhead is low for sets with a few functions per task and/or with a small number of tasks.

Unsurprisingly, design points with low overhead tend to also be the ones with a higher rate of schedulability. The lower the overhead, the closer the protected task set is to the initial and unprotected set and the higher its probability of being schedulable.

When we compare the distribution of the overhead with and without task pushing, we make two observations. First, 73% of the labeled sets have less than 5% of overhead, compared to 69% without task pushing. Task pushing reduces the average overhead of 230 different task set design points to be under the limit of acceptable overhead for industrial embedded system security solutions [16]. Second the worst cases of task sets, which have above 60% of overhead with all tasks instrumented, merely reach a 40% slowdown after the tasks are pushed.

The RECFISH overhead depends a lot on which class of task sets we are examining and it is the same for task pushing: it works well on some and may not change anything in others. We found in our examination that there are a few design points for which none of the 10 generated task sets situation could be improved by task pushing, while another saw its overhead reduced from 70% to less than 2%. Such a reduction can be explained by the fact that this design points regroups task sets with a lot of branches and functions per tasks (therefore, a large overhead without task pushing) but also larges period and low execution times. In this particular case, task pushing is extremely effective.

5.4 Result of Under-approximation of Schedulability for Hybrid EDF Scheduler

With the under-approximation of the time demand for each task during their extra time window (see Section 4.3), we can simulate the execution only on a period of the longest task. We use a queue to store the task calls and the end of executions of the current job of each task. Initially, the queue is filled with all the calls of each task in this period (the tasks are called so that they all finish one period at the end of the time window, see Figure 3), plus the end of executions of the tasks with an extra time window. Then for each step of the simulation, we pick a task to run and simulate its execution by delaying the execution of each other task until the next scheduling event (end of execution of the running task or task call). We run this simulation until a job misses its deadline or we reach the end of the time window with each of the job's execution finished. In the former case the set is registered as unschedulable, in the latter it is registered as schedulable. Using the exact same task set generator as for the fixed priority scheduler evaluation, we run this test for 100 samples for each of the 5,760 design points.

As a result, we find that this approximation finds only 52% of schedulable sets where the earliest deadline first policy utilization feasibility test gives a result of 100% (the set generator only create task sets with a total utilization not greater than 100%). This under-approximation is not accurate enough to be applied to the hybrid scheduler. To increase this accuracy, we run the same test on a larger number of periods. Running it on 10 periods only increases the accuracy of the test only to 59% while increasing the computation time by an order of magnitude. The computation time will skyrocket if we plan to increase the number of periods in order to reach an acceptable accuracy, which makes it unpractical.

To improve this feasibility test, we can search for a better way to simulate the execution of the task during their extra time window. If we manage to have a more precise idea of how much time is used by each task without having to simulate too many periods, the accuracy will improve. It would also be possible to conduct the same simulation but in decreasing time: we start the simulation at $t = H$ and run backward until there is an inconsistency (a job needs to run before it is called to meet its deadline) or an idle time. This way we ensure an exact simulation without having to always simulate during a window of length H. Another path we can explore is to totally change the type of test we are looking for. For a set to be schedulable in a hybrid scheduler, non-critical tasks must not be delayed more than their minimum slack time (*i.e.*, the minimum amount of time between their end of execution and their deadline). If this measure is provided, we can use it

to ensure that the set is still schedulable after adding the critical tasks, which delay all the non-critical ones by the same amount of time. To the best of our knowledge, we do not know if such a tool exists.

6 Conclusion

We showed that task pushing can be an efficient way to reduce the overhead of instrumentation based defenses for fixed priority schedulers. We assumed in this work that we know a priori if tasks are tainted and did not cover how to determine these labels. We would need a static analysis pass to implement this classification. Alias analysis [14] seems to be a promising approach. By listing aliases of user inputs, we may be able to list and taint all tasks that use these variables for computation. We would also relax some other assumptions with how the tainted tags spread from one task to another. In our definition, every task which receives data from a tainted task must be tagged as tainted. Nevertheless, we may be able to guarantee statically that communication is safe by analyzing the type of data exchanged. For instance, we may not record a task as tainted if the only variable it receives from a tainted task is an `int` variable due to its fixed size. Therefore, type-based analysis [13] may be an effective tool to gain a more precise definition of the taint label.

Second, we could not apply the idea of task pushing directly to the earliest deadline first scheduler. We proposed a simple modification of the policy that would allow pushing tasks to reduce the negative impact the defenses, but we have not found yet a practical way to test the schedulability of task sets on this new scheduler. We showed that a direct simulation of the execution is not a practical solution and proposed an under-approximation, which is not accurate enough. We will have to find a suitable schedulability test for this new scheduler in order to measure the effectiveness of task pushing on the hybrid scheduler.

References

[1] FreeRTOS project presentation website. `http://www.freertos.org`.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[3] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Real-Time Systems Symposium, 11th*, [1990] Proceedings 11th Real-Time Systems Symposium, pages 182,190. IEEE Computer Society Press, 1990.

[4] B. Brandenburg. Schedcat. `https://github.com/brandenburg/schedcat`.

[5] B. Brandenburg. *Scheduling and Locking Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[6] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. Formatguard: Automatic protection from printf format string vulnerabilities. In *USENIX Security Symposium*, volume 91. Washington, DC, 2001.

[7] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguardtm: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium*, volume 12, pages 91–104, 2003.

[8] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[9] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88. USENIX Association, 2006.

[10] D.-Z. He, F.-Y. Wang, W. Li, and X.-W. Zhang. Hybrid earliest deadline first /preemption threshold scheduling for real-time systems. In *Proceedings of 2004 International Conference on Machine Learning and Cybernetics (IEEE Cat. No.04EX826)*. IEEE, 2004.

[11] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[12] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[13] J. Palsberg. Type-based analysis and applications. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 20–27. ACM, 2001.

[14] E. Ruf. Context-insensitive alias aliasing reconsidered. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 13–22. ACM, 1995.

[15] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[16] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[17] R. J. Walls, N. F. Brown, T. Le Baron, C. A. Shue, H. Okhravi, and B. Ward. Control flow integrity for real-time embedded systems. In *ECRTS*, 2019.