Minimizing Side-Channel Attack Vulnerability via Schedule Randomization

Nils Vreman

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6059
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Printed in Sweden by Tryckeriet i E-huset
Lund 2018

Abstract

Predictable and repeatable execution is the key to ensuring functional correctness for real-time systems. Scheduling algorithms are designed to generate schedules that repeat after a certain amount of time has passed. However, this repeatability is also a vulnerability when side-channel attacks are considered.

Side-channel attacks are attacks based on information gained from the implementation of a system, rather than on weaknesses in the algorithm. Side-channel attacks have exploited the predictability of real-time systems to disrupt their correct behavior.

Schedule Randomization has been proposed as a way to mitigate this problem. Online, the scheduler selects a schedule among a set of available ones, trying to achieve an execution trace that is as different as possible from previous ones, therefore minimizing the amount of information that the attacker can gather.

This thesis investigates fundamental limitations of schedule randomization for a generic taskset. We then propose an algorithm to construct a set of schedules that achieves a differentation level as high as possible, using the fewest number of schedules, for tasksets with implicit deadlines. The approach is validated with synthetically generated tasksets and the taskset of an industrial case study, showing promising results.

Acknowledgements

To everyone believing in me, supporting me, and encouraging me I dedicate this master's thesis; I owe it all to you!

To my loving mother, father, and brother (Tine, Wim and Kalle); Thank you for being there for me when I needed you the most and that you always had my back. This master's thesis would not have been possible without your everlasting support.

I am very grateful to have had Martina Maggio as my supervisor, cheering me on and coaching me throughout this thesis. With her support, I have gained more confidence in myself as well as more curiosity for new and interesting problems.

I would like to thank Richard Pates for all the help he has given to me. Without him, the main contribution of this thesis would still be left unproven. Also, I am very grateful for him giving me advice on how to approach new problems.

A special thanks goes out to Carl Nilsson who has always been there for me to share my tears and laughs and to provide interesting discussions. I am also thankful for all of my other friends who have supported me throughout my journey.

Finally, to all the wonderful people at the department of Automatic Control, thank you for making me feel included and for sharing your knowledge with me. I have thoroughly enjoyed every minute of my time here.

Contents

1. Introduction **9**
- 1.1 Background . . . 9
- 1.2 Side-Channel Attacks . . . 10
- 1.3 Related Work . . . 11
- 1.4 Goals and Contribution . . . 11
- 1.5 Outline . . . 12

2. Side-Channel Attacks **13**
- 2.1 Motivation and Classification . . . 13
- 2.2 Attacks . . . 14

3. Problem **16**
- 3.1 Problem Description . . . 16
- 3.2 Example . . . 19
- 3.3 Problem Formulation . . . 20

4. Analytical Investigation **21**
- 4.1 Fundamental Limits . . . 21
- 4.2 Example . . . 25

5. Schedule Set Generation for Implicit Deadline Tasksets **26**
- 5.1 Notation . . . 27
- 5.2 Fundamental Algorithm Idea . . . 27
- 5.3 In Depth . . . 28

6. Experimental Results **33**
- 6.1 Synthetic Tasksets . . . 33
- 6.2 Case Study . . . 36

7. Conclusion and Future Work **38**
- 7.1 Summary . . . 38
- 7.2 Future Work . . . 38
- 7.3 Conclusion . . . 40

Bibliography **41**

1

Introduction

This chapter introduces the context for this work and presents some background and motivation.

1.1 Background

The set of operations to be executed on an embedded device, for example the operations required to run a controller, is often partitioned into tasks which run concurrently on the hardware. These tasks are assigned attributes, such as execution time, period, and deadline. Furthermore, it is essential that the tasks are given computational time equal to their execution time in each time span from their activation to their deadline. If there exists a resource distribution such that all tasks meet their corresponding deadline, the taskset is called schedulable [Årzén, 2014, pp. 145].

Given any schedulable taskset, a schedule could be generated such that each task is allocated its relative portion of the computational resources. Utilizing such a schedule has the advantage of the tasks being scheduled deterministically, assuring each deadline will be met. However, there are disadvantages to the scheduler being deterministic as well. Every hyperperiod (the least common multiple of all the task periods in the taskset) the scheduler will repeat itself and is therefore predictable and vulnerable to timing inference attacks. Such an attack aims to abuse the temporal execution of the embedded device to either disrupt the normal behaviour or to extract information from the device. For instance, if a taskset is transmitting data at a recurring time interval, disrupting the transmission during this time interval would result in data being lost. This disruption could be done surreptitiously to avoid detection from any security system, since a detection task running on the hardware would be unaware that something has happened. To mitigate this problem, schedule randomization was proposed.

Given a taskset, a set of schedules (unrelated to one another) could be generated such that each schedule is valid, i.e. all the tasks meet their deadlines.

To quantify the randomness of such a schedule set, an entropy-like notion was introduced. Schedule set "entropy", closely related to physical entropy, quantifies the diversity among schedules in the set. A natural question then arises: Is there a finite upper bound on the schedule set entropy accounting for the temporal execution constraints?

Another problem arises from the fact that embedded devices are often very limited by their memory size. It is therefore crucial that the size of the schedule set is minimized.

Generating a schedule set with "entropy" as high as possible using as few schedules as possible is the goal of this thesis.

1.2 Side-Channel Attacks

In a world with rapidly growing interest in embedded systems, e.g. Internet of Things, cars, medical equipment, cell phones etc., the need for safe real-time systems is greater than ever. The growing number of mobile decives have caused an increase in side-channel attacks aimed toward these [Spreitzer et al., 2017]. It is essential that these devices are safe to use because errors or disruptions in the normal task executions might have dire consequences.

Generic side-channel attacks can be classified as all attacks exploiting information about the system rather than bugs and vulnerabilities in the code itself. For example, execution information from the scheduler can in the hands of an attacker impose serious danger on leakage of information or disruption of certain tasks. In [Spreitzer et al., 2017] the authors state that information from a system can be surreptitiously extracted by observing power usage, execution time, cached memory etc. Traditionally these attacks have been somewhat troublesome to perform for the attacker, since the attacker had to be located near (Local side-channel attacks) or close by (Vicinity side-channel attacks) to the victim, e.g., jamming the transmission to or from the system using electromagnetic fault injections [Spreitzer et al., 2017]. However, the era of cloud computing connected most embedded systems to the internet and thereby also expanded the scope of side-channel attacks such that they could be performed remotely (Remote side-channel attacks). Taking remote control of an embedded device's sensors could for example grant you system information by exploiting your knowledge about the power usage.

There is an urgent need for counter-measures due to the sheer number of ways to attack these devices. Given this threat, introducing randomization into an embedded device's scheduler might increase the resistance against side-channel attacks. This master's thesis aims at impeding side-channel attacks on embedded systems.

1.3 Related Work

Security in real-time embedded systems has been an increasingly popular topic in the last couple of years. In [Jiang et al., 2014] the authors discuss the pre-existing security in pre-emptive earliest-deadline first (EDF) schedulers and pre-emptive rate-monotonic scheduling (RMS) against differential power analysis attacks (DPAs). With the attacker model used in the paper, the execution time of tasks is assumed unknown and therefore introduces a "random" aspect which helps prevent side-channel attacks. Another related paper is [Spreitzer et al., 2017] in which the authors discuss and classify different types of side-channel attacks in the form of a case-study. The authors are not presenting any solutions but their classification system and discussion is closely related to what this thesis is trying to solve.

In [Chen et al., 2017] the authors present an algorithm for attacking and extracting the exact schedule run on a real-time embedded system. With a high success rate they manage to extract the entire schedule from a fixed-priority pre-emptive hard real-time system by abusing the periodicity of the system. This paper gives us insight, regarding how an attacker could try to exploit the embedded device.

The paper by [Yoon et al., 2016] presents an alternative solution to the problem presented in this thesis. The authors of that paper introduce the "upper-approximated entropy", and they present an online based solution rather than an offline based one. This reports expands their entropy-like notion by introducing bounds and mathematical proofs as well as an optimal solution.

1.4 Goals and Contribution

Previous attempts to introduce randomization in real-time embedded systems scheduling [Yoon et al., 2016; Kruger et al., 2018] have used an algorithmic approach and tried to compute — in the shortest amount of time possible — randomized version of the schedule that would strive to maximize the upper-approximated entropy of the list of schedules used at runtime. This work takes a complementary approach.

Rather than focusing on the schedule set generation, we first focus on analytical fundamental limitations of the problem. We analyzed the problem and found out that we can safely compute an upper bound to the achievable upper-approximated entropy, taking into account the constraints introduced by the taskset characteristics. We then analyzed the number of schedules that we need to introduce in our schedule set to achieve said upper bound.

For a subset of the possible tasksets — i.e., for the tasksets where for each task, the activation period is equal to the computation deadline — we

have designed an algorithm that produces a schedule set that reaches both the mentioned bounds — i.e., an algorithm that uses the minimum number of schedules to achieve the maximum upper-approximated entropy possible. Finally, we evaluated our algorithm with synthetically generated tasksets, as well as for an industrial case study.

1.5 Outline

The thesis starts with introducing side-channel attacks and their categories in Chapter 2 and continues by presenting the problem formulation in depth in Chapter 3. Alongside the problem formulation an example is presented, which will be used to explain new theory.

Following the problem formulation, Chapter 4, new theorems and definitions are presented and proven. The analytical results acquired here will lay ground to the following chapters. For example, in Chapter 5 we present the algorithm which has been designed to solve the problem formulated prior.

Results acquired from running a set of benchmark tests are presented in Chapter 6. The accurracy of the algorithm as well as graphs and plots of the results are presented here. A discussion about why the results behave as they do is also provided.

The final chapter (Chapter 7) concludes the report with possible future research extensions, obstacles faced along the course of the master's thesis, and conclusions drawn from the analytical and algorithmic results.

2

Side-Channel Attacks

In this chapter we introduce the motivation behind why security measures are needed in real-time schedules. We describe a classification system, introduced in [Spreitzer et al., 2017], that classifies different side-channel attacks and analyzes their scope and scale. A few attack examples are also presented to clarify what type of attacks we are trying to prevent, and why it is so important to prevent them.

2.1 Motivation and Classification

[Spreitzer et al., 2017] introduces a classification system of side-channel attacks. The proposed categorization system uses three attributes.

1. *Passive* vs *active*:
   An attacker who is *passive* observes the system without interfering whereas an *active* attacker modifies the device and/or its behaviour.

2. *Physical properties* vs *logical properties*:
   This attribute specifies what kind of property the attacker targets with his attack. An attacker could target hardware specific information (*physical properties*), such as power consumption, sensor data, etc., or software specific information (*logical properties*), such as memory footprint, cached data, etc.

3. *Logical attackers* vs *vicinity attackers* vs *remote attackers*:
   Depending on the location of the attacker, the attack classification is split even further. *Local attackers* must be so close to the device that they can access it directly. *Vicinity attackers* needs to be close enough to be able to observe and listen in on the network the device is accessing. Lastly, *remote attackers* are only required to run software on the device or for the device to access a remote server (website or cloud).

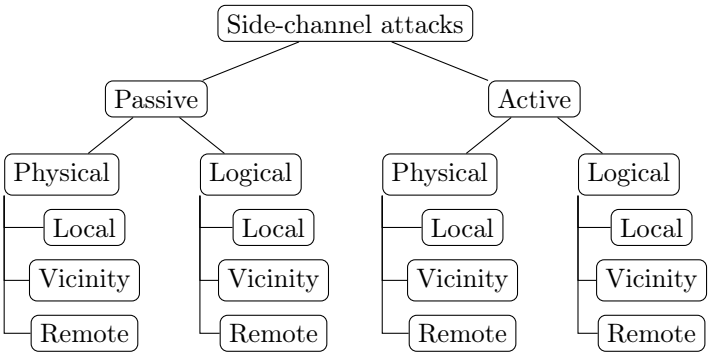


Figure 2.1 A visualization of the categorization system created by [Spreitzer et al., 2017].

This categorization system (seen in Figure 2.1) is used to understand specific attacks and to analyze what could be done to prevent them. Information about the possible attackers an embedded device could encounter helps us develop countermeasures for certain classes of attackers. Hence, the categorization system is used to support the identification of threats, and to support the methods developed to prevent those threats.

2.2 Attacks

There are many side-channel attacks. In this section we will present a few attacks, that our proposed solution (Chapters 4 and 5) helps preventing.

In [Pietro et al., 2014] the authors discuss different security problems in wireless ad-hoc networks, e.g., communicating autonomous cars. A wireless network is constructed of nodes. One problem arises when an attacker tries to tamper with the sensors in one of the nodes and replace them with malicious sensors under the control of the attacker. This could result in the attacker sending erroneous data, stealing data, or shutting down the sensor, which could all result in hazardous outcomes.

Further research has been done in the area of security for wireless ad-hoc networks for autonomous cars. A survey authored by [Mejri et al., 2014] presents similar conclusions to the survey by [Pietro et al., 2014]. In the survey they claim that jamming attacks are problematic in the sense that they work as a physical Denial-of-Service (DoS) attack. In other words, an attacker could transmit a signal at a specific time interval to disrupt the communication channel. This could render the node isolated and thereby cause major harm to surrounding nodes. Both of these attacks try to *actively* target *physical properties* of the device. Since, the attacker has to be part

of, or has access to, the network these attacks would be classified as *vicinity* attacks.

The next attack we discuss is the *ScheduLeak* algorithm [Chen et al., 2017]. The goal of the *ScheduLeak* algorithm is to *leak* the entire *schedule* a device uses. When an attacker has the schedule, then for example DoS attacks (as discussed earlier [Mejri et al., 2014]) are easier to carry out, and sensors are easier to replace. To extract the full scheudle the authors of [Chen et al., 2017] assume an attacker model where the taskset information is known and that the attacker has the opportunity to run one or more tasks in the system if needed. However, the arrival time of each task is assumed unknown. With the authors's proposed algorithm, they can extract the entire schedule from a running system just by observing the *busy intervals* (an interval where tasks are executing) of the schedule. Would an attacker acquire the exact schedule of a system, he could disrupt certain tasks covertly. Classifying the attacker is now simple: Since the attacker's objective is to observe the system and extract the schedule, it is a *passive* attack. It interacts with *physical* attributes such as timing information. Finally, the *ScheduLeak* algorithm can be executed from anywhere, resulting it being a *remote attack*.

Another side-channel attack is the differential power analysis (DPA), as defined by [Kocher et al., 1999]. DPA uses the power trace from processed data to extract cryptographic keys. To extract these keys, the attack relies heavily on a stochastic evaluation of the recurring power trace the attacker can acquire from the system. The authors propose a few solutions to mitigate this attack, one of them being temporal obfuscation. Since the attacker has to be in physical possession of the device and only extracts information about the physical properties without interfering with the execution, the attack could be classified as a passive, physical, and local side-channel attack.

A similar attack to the DPA is the differential computation analysis (DCA) attack [Bos et al., 2016]. The authors present this modern version of the DPA which targets white-box crypto implementations, a way of embedding the cryptographic key into the software implementation. In their attack model they assume the attacker has complete control over the device (not that uncommon in today's world of smart cell phone apps). They use dynamic binary instrumentation (commonly used to "allow one to monitor, modify and insert instructions in a binary executable" [Bos et al., 2016]) to precisely monitor the execution of the program and then use DPA to extract the crypto key. As well as the authors of [Kocher et al., 1999], the authors of [Bos et al., 2016] propose the use of temporal obfuscation to mitigate the threat of DCA attacks.

3

Problem

In this chapter we formalize the problem of randomizing the execution of a schedulable taskset (composed of periodic tasks) in a platform with limited storage space. The problem comes from the desire to avoid side-channel attacks, i.e., attacks in which an attacker can learn (and exploit) the behavior of the system by observing its execution for a certain amount of time.

3.1 Problem Description

A taskset $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_m\}$ of m independent, periodic tasks is given. Each task τ_i is defined as the tuple $\tau_i = \{e_i, t_i, d_i\}$, where e_i is the execution time to be given to the task, t_i is the task activation period, and d_i is the task deadline. We denote with U the taskset utilization, i.e., $U = \sum_{i=1}^{m} e_i/t_i$. We assume that the taskset is schedulable; ergo, there exists a resource distribution such that each task τ_i meets its corresponding deadline [Årzén, 2014, pp. 145-146]. We denote with ℓ the hyper-period of the taskset, i.e., the least common multiple $\mathrm{lcm}(\cdot)$ of the task periods, $\ell = \mathrm{lcm}(t_1, \ldots, t_m)$.

Classical scheduling algorithms such as earliest deadline first (EDF) and fixed priority (FP) are applicable in both periodic task models as well as models where the tasks are aperiodic or sporadic. We assume a task model utilizing static-cyclic scheduling, i.e., the task model does not consider unknown or unplanned interrupts [Årzén, 2014].

A schedule s for the taskset is a sequence of ℓ elements, that contains numbers in the set $\{0, 1, \ldots, m\}$. Formally,

$$s = (s_1,\ s_2,\ \ldots,\ s_\ell)\,;\ s_j \in \{0, 1, \ldots, m\}. \tag{3.1}$$

We denote with s_j the value of the element in position j, i.e., the task that is executed according to the schedule s in the j-th time unit. If the element is a zero, the idle task is executed. If the element is a positive number i, then τ_i is executed. Given that the taskset $\mathcal{T}$ is schedulable, we can safely assume that s respects all the constraints that for each task and each activation,

the schedule assigns the prescribed execution time before the corresponding deadline.

It is our objective to determine a set $\mathcal{K}$ of k schedules that is as diverse as possible, keeping k as small as possible. The first part of the problem is determining the set of all the schedules that satisfy the constraints on execution times, periods, and deadlines. This set in principle can have high cardinality, and it may not be possible to store all the schedules in the memory of the execution platform, e.g., an embedded system may have limited storage for alternative schedules. Suppose we obtain a set of n valid schedules $\mathcal{S}$ for the given taskset (each of length ℓ), $\mathcal{S} = \{s^{(1)}, s^{(2)}, \ldots, s^{(n)}\}$. The second part of the problem is then to select k schedules (from the set $\mathcal{S}$) to include in the set $\mathcal{K}$, to maximize the *diversity* of the schedules and avoid the possibility that the attacker gathers information about the tasks execution.

The diversity of the set of selected schedules $\mathcal{K}$ can be measured in different ways. The authors of [Yoon et al., 2016] propose the use of entropy-like metrics, the *slot entropy* and the *upper-approximated entropy*. In the following part we first define the slot count $C_{j,i,\mathcal{K}}$, as the number of occurrences of a task i in a given scheduling slot j in the set $\mathcal{K}$, and then use that to formally define the slot entropy $H_j(\mathcal{K})$ and the upper-approximated entropy $\tilde{H}(\mathcal{K})$. We denote with $\phi(x)$ the function

$$\phi(x) = \begin{cases} 0 & x \leq 0 \\ -x \cdot \log_2(x) & x > 0 \end{cases}.$$

$\phi(x)$ is the summand function of the Shannon entropy function $H(x) = \sum_i \phi(x_i)$ [Shannon, 1948]. $\phi(x)$ can also be seen in Figure 3.1.

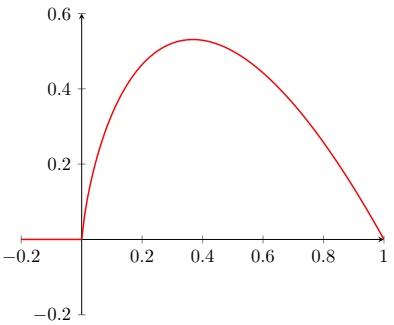


Figure 3.1 A visualization of the Shannon entropy summand.

Definition 1
Given a set of k valid schedules $\mathcal{K} = \{s^{(1)}, s^{(2)}, \ldots, s^{(k)}\}$ for the taskset $\mathcal{T}$, the j-th time unit, and the i-th task τ_i, we define the slot count $C_{j,i,\mathcal{K}}$ as a

function that counts the occurrences of the task i in the given position j in the set $\mathcal{K}$. Using the square brackets as the Iverson brackets — that evaluates to 1 if the proposition inside the bracket is true, and to 0 otherwise — we can then write $C_{j,i,\mathcal{K}}$ as

$$C_{j,i,\mathcal{K}} = \sum_{s^{(q)} \in \mathcal{K}} \left[s_j^{(q)} = i \right] = \sum_{q=1}^{k} \left[s_j^{(q)} = i \right]. \tag{3.2}$$
□

Using the slot count, we can now formally write the slot entropy and the upper-approximated entropy according to the definitions given in [Yoon et al., 2016].

Definition 2
The *slot entropy* $H_j(\mathcal{K})$ can be written as a function of the tasks found in slot j, i.e.,

$$H_j(\mathcal{K}) = \sum_{i=0}^{m} \phi\left(\frac{C_{j,i,\mathcal{K}}}{k}\right) = \sum_{i=0}^{m} -\frac{C_{j,i,\mathcal{K}}}{k} \cdot \log_2 \frac{C_{j,i,\mathcal{K}}}{k} \tag{3.3}$$
□

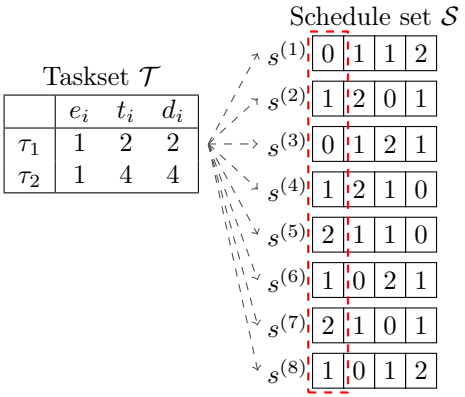
Definition 3
The *upper-approximated entropy* is the sum of all the slot entropies in the hyper-period, i.e.,

$$\tilde{H}(\mathcal{K}) = \sum_{j=1}^{\ell} H_j(\mathcal{K}) = \sum_{j=1}^{\ell} \sum_{i=0}^{m} -\frac{C_{j,i,\mathcal{K}}}{k} \cdot \log_2 \frac{C_{j,i,\mathcal{K}}}{k} \tag{3.4}$$
□

A short explanation of each definition is given. Since Definition 1 solely defines a variable $C_{j,i,\mathcal{K}}$ which counts occurrences of τ_i at time slot j, it is left self-explained.

Definition 2 however might not be quite as self-explanatory. First we highlight the expression $C_{j,i,\mathcal{K}}/k$ that is frequently occurring. This expression can be seen as the relative occurrence (or probability) of task τ_i at time slot j in the set $\mathcal{K}$. Since the cardinality is $|\mathcal{K}| = k$ we get the relative frequency of task τ_i at time slot j as the occurrence rate $C_{j,i,\mathcal{K}}/k$. Secondly, the definition in [Yoon et al., 2016] is based on Shannon entropy from information theory which is used to measure the stochasticity of data. This is what is used together with the relative frequency to give the expression in Definition 2.

Finally, Definition 3, summing up all of the slot entropies gives us a quantity we can compare to other schedule sets as well as give us an idea of how diverse the schedule set is. A higher value of the upper-approximated entropy results in a more diverse schedule set.



$$
\begin{aligned}
H_1(\mathcal{S}) &= -\frac{C_{1,0,\mathcal{S}}}{n} \cdot \log_2 \frac{C_{1,0,\mathcal{S}}}{n} \\
&\quad -\frac{C_{1,1,\mathcal{S}}}{n} \cdot \log_2 \frac{C_{1,1,\mathcal{S}}}{n} \\
&\quad -\frac{C_{1,2,\mathcal{S}}}{n} \cdot \log_2 \frac{C_{1,2,\mathcal{S}}}{n} \\
&= -\tfrac{2}{8} \cdot \log_2 \tfrac{2}{8} - \tfrac{4}{8} \cdot \log_2 \tfrac{4}{8} - \tfrac{2}{8} \cdot \log_2 \tfrac{2}{8} \\
&= -\tfrac{1}{2} \cdot \log_2 \tfrac{1}{4} - \tfrac{1}{2} \cdot \log_2 \tfrac{1}{2} = 1.5
\end{aligned}
$$

Figure 3.2 Example of slot entropy computation for a given taskset $\mathcal{T}$ and schedule set $\mathcal{S}$.

3.2 Example

Presented below is a very simple example to help visualizing the quantities just introduced. This example will be used continuously in the remainder of this thesis.

Suppose we have a taskset $\mathcal{T}$ composed of two different tasks, $\mathcal{T} = \{\tau_1, \tau_2\}$. The first task τ_1 has execution time $e_1 = 1$, and period and deadline $p_1 = d_1 = 2$. The second task has execution time $e_2 = 1$ and period and deadline $p_2 = d_2 = 4$. The hyper-period is then $\ell = 4$. The list of valid schedules for this taskset comprises 8 schedules, $\mathcal{S} = \{s^{(1)}, \ldots, s^{(8)}\}$.

Figure 3.2 shows the sequences and the computation of the slot entropy for the entire set $\mathcal{S}$ and the first slot $j = 1$, $H_1(\mathcal{S}) = 1.5$. With analogous calculations, it can be shown that the slot entropy for the other slots is the same. The upper-approximated entropy for the set $\mathcal{S}$ is then $\tilde{H}(\mathcal{S}) = 4 \cdot 1.5 = 6$. Computing the power set of $\mathcal{S}$ and the upper-approximated entropy for each of the elements of the power set (the set containing all subsets to $\mathcal{S}$), it is possible to show that for $k = 2$, the maximum upper-approximated entropy achievable is 4. With $k = 3$ it is possible to reach an upper-approximated entropy ≈ 5. For $k = 4$ (and $k = 8$) the maximum of 6 is reached. With $k = 5$, $k = 6$, and $k = 7$ it is possible to respectively reach values ≈ 5.786, ≈ 5.837,

and ≈ 5.871.

In this case then, the minimal set of schedules that achieves the highest upper-approximated entropy has cardinality $k = 4$. One possible optimal set is $\mathcal{K} = \{s^{(1)}, s^{(2)}, s^{(5)}, s^{(6)}\} = \{(0\,1\,1\,2), (1\,2\,0\,1), (2\,1\,1\,0), (1\,0\,2\,1)\}$. The 4-elements set $\mathcal{K}$ that gives the maximum upper-approximated entropy is not unique. Another alternative would be $\mathcal{K} = \{s^{(3)}, s^{(4)}, s^{(7)}, s^{(8)}\}$. If the example is as simple as this one, it is possible to enumerate all the alternatives and determine optimal solutions with a brute force approach. However, for tasksets with large hyper-periods, it is infeasible to enumerate all the valid schedules and to compute the power set of these schedules to determine an optimal solution.

3.3 Problem Formulation

It is now possible to formally state the objective of minimizing the risk that an attacker would be able to extract and effectively use information to compromise the system. Given a taskset $\mathcal{T}$ and the set of all the valid schedules $\mathcal{S}$, what is the smallest subset of $\mathcal{S}$ that maximises the upper-approximated entropy?

Mathematically, this problem can be written as the following optimization problem.

Problem 1
Given a taskset $\mathcal{T}$ and a valid set of schedules $\mathcal{S}$, solve

$$\begin{aligned} \mathcal{K}^* &= \arg\min_{\mathcal{K}\subseteq\mathcal{S}} |\mathcal{K}| \\ \text{s.t. } \tilde{H}(\mathcal{K}) &= \max_{\mathcal{L}\subseteq\mathcal{S}} \tilde{H}(\mathcal{L}). \end{aligned}$$

□

Here $\mathcal{L}$ is any generic subset of $\mathcal{S}$. We are searching for the set $\mathcal{K}$ with the minimum cardinality that achieves the maximum upper-approximated entropy. This problem consists of two main aspects. First we must determine the maximum upper-approximated entropy; that is we must solve

$$\tilde{H}^* = \max_{\mathcal{L}\subseteq\mathcal{S}} \tilde{H}(\mathcal{L}).$$

Then we must find the smallest subset $\mathcal{K}^* \subseteq \mathcal{S}$ with upper-approximated entropy $\tilde{H}(\mathcal{K}^*) = \tilde{H}^*$.

4

Analytical Investigation

In this chapter we show that the properties of the taskset $\mathcal{T}$ create fundamental limits both on $\tilde{H}^*$, and on the cardinality of the optimal subsets $\mathcal{K}^* \subseteq \mathcal{S}$. These limits will help us exclude a lot of subsets, hence pruning the search space for an optimal schedule set $\mathcal{K}^*$. Practically this will give us the dimensions and constraints on the schedule set, making it possible to generate in a feasible amount of time.

4.1 Fundamental Limits

Problem 1 could be approached by an exhaustive search. If one calculated the upper-approximated entropy for every element in the power set of $\mathcal{S}$, the optimal solution to Problem 1 could be obtained by choosing the smallest subset that achieved $\tilde{H}^*$. However since the cardinality of $\mathcal{S}$ is typically large, this will be infeasible in practice. Such an approach is also naïve. After all it seems improbable that the solution to Problem 1 would have cardinality one, so we could rule those subsets out of our exhaustive search.

One natural questions then arises: Are there any fundamental limits on the achievable maximum upper-approximated entropy or the cardinality of $\mathcal{K}$? The remainder of this section is devoted to answering this question. We show that the properties of the taskset $\mathcal{T}$ impose fundamental limits both on $\tilde{H}^*$, and on the cardinality of the subsets that can achieve this bound.

Entropy Bound

The fact that each task in the taskset $\mathcal{T}$ must be executed with a certain frequency imposes a fundamental limit on the maximum upper-approximated entropy. The intuitive explanation for this is as follows: Our objective is to select a set of schedules that minimizes the information an attacker can obtain by observing the execution of any individual task. We therefore want it to appear as if each task was allocated randomly to each given slot. However we cannot necessarily make this allocation appear random with equal probability

(for each task), because we are required to execute tasks for given time units a certain number times in each hyper-period. Therefore the best we can do is make each task appear with probability specified by its relative frequency in the hyper-period. This relative frequency can be seen as the utilization $u_i = {}^{e_i}/_{t_i}$ of task τ_i. The entropy of the corresponding random variable then specifies an upper bound on the upper-approximated entropy of any set of schedules $\mathcal{K} \subseteq \mathcal{S}$.

THEOREM 1
Given any schedule set $\mathcal{K} \subseteq \mathcal{S}$, with $|\mathcal{K}| = k$,

$$\tilde{H}(\mathcal{K}) \leq \ell \cdot \sum_{i=0}^{m} \phi\left({}^{e_i}/_{t_i}\right).$$

Proof When $x > 0$, the function $\phi(x) = -x \cdot \log_2(x)$ is continuous and concave. The concavity implies that ${}^1/_p \sum_{i=1}^{p} \phi(x_i) \leq \phi\left({}^1/_p \sum_{i=1}^{p} x_i\right)$.

Applying this inequality to the expression in Equation (3.4), we obtain

$$\frac{1}{\ell} \sum_{j=1}^{\ell} \sum_{i=0}^{m} \phi\left(\frac{C_{j,i,\mathcal{K}}}{k}\right) \leq \sum_{i=0}^{m} \phi\left(\frac{1}{\ell} \sum_{j=1}^{\ell} \frac{C_{j,i,\mathcal{K}}}{k}\right),$$

where the expression $\sum_{j=1}^{\ell} \frac{C_{j,i,\mathcal{K}}}{k}$ counts – for all the schedules in $\mathcal{K}$ – the amount of occurrences of each task in the hyper-period, and can be written as

$$\sum_{j=1}^{\ell} \frac{C_{j,i,\mathcal{K}}}{k} = \ell \cdot \left(\frac{k \cdot e_i}{t_i} \cdot \frac{1}{k}\right) = \ell \cdot \frac{e_i}{t_i},$$

thus leading to

$$\frac{\tilde{H}(\mathcal{K})}{\ell} = \frac{1}{\ell} \sum_{j=1}^{\ell} \sum_{i=0}^{m} \phi\left(\frac{C_{j,i,\mathcal{K}}}{k}\right) \leq \sum_{i=0}^{m} \phi\left(\frac{\ell}{\ell} \frac{e_i}{t_i}\right) = \sum_{i=0}^{m} \phi\left(\frac{e_i}{t_i}\right).$$

Knowing that $\ell > 0$, we then derive the upper bound

$$\tilde{H}(\mathcal{K}) \leq \ell \cdot \sum_{i=0}^{m} \phi\left({}^{e_i}/_{t_i}\right) = \tilde{H}^{ub}, \tag{4.1}$$

for the achievable upper-approximated entropy. □

The bound in Equation (4.1) is denoted with $\tilde{H}^{ub}$ for simplicity reasons. Two additional results can be found following the same methodology. The first one is a more relaxed bound (i.e., a quantity that upper bounds $\tilde{H}^{ub}$). The second, on the contrary, tightens $\tilde{H}^{ub}$ when at least for one task i in the taskset the period t_i is not equal to the deadline d_i. This second bound is equivalent to $\tilde{H}^{ub}$ when all the deadlines are equal to the periods.

Corollary 1
Given any schedule set $\mathcal{K} \subseteq \mathcal{S}$,

$$\tilde{H}(\mathcal{K}) \leq \ell \cdot \sum_{i=0}^{m} \phi\left({}^{e_i}/_{t_i}\right) \leq -\ell \cdot log_2\left({}^{1}/_{(1+m)}\right) = \ell \cdot log_2\left(1+m\right). \tag{4.2}$$

□

This bound directly comes from applying the same principle used to prove Theorem 1 to Equation (4.1). The expression in Equation (4.2) is not always reachable, depending on the characteristics of the taskset. This leads us to consider the taskset characteristics. In particular, we can compute a bound that takes into account the utilization U of the taskset, leading to the following corollary.

Corollary 2
Given any schedule set $\mathcal{K} \subseteq \mathcal{S}$,

$$\tilde{H}(\mathcal{K}) \leq \ell \cdot \{-(1-U) \cdot log_2(1-U) - U \cdot log_2({}^{U}/_{m})\}. \tag{4.3}$$

□

The contribution of the idle task to the upper-approximated entropy is equal to $\ell \cdot \{-(1-U) \cdot log_2(1-U)\}$, recall that $1 = \sum_{i=0}^{m} {}^{e_i}/_{t_i} = U + {}^{e_0}/_{t_0}$. The maximum value for the upper-approximated entropy is reached when the utilizations of the tasks allow them to be evenly distributed. The contribution of each of them is then $-\ell \cdot {}^{U}/_{m} \cdot log_2({}^{U}/_{m})$. Deriving the expression in Equation (4.3) allows us to also study when we can be closer to the bound in Equation (4.2) — and when we can expect to reach the maximum upper-approximated entropy reachable for a set of m tasks, depending on the task characteristics. With respect to the utilization $U = \sum_{i=1}^{m} {}^{e_i}/_{t_i}$, the upper-approximated entropy can reach its maximum when the utilization of the system is equal to $U = {}^{m}/_{(1+m)}$.

Corollary 3
Given any schedule set $\mathcal{K} \subseteq \mathcal{S}$,

$$\tilde{H}(\mathcal{K}) \leq \sum_{j=1}^{\ell} \phi\left(1-U\right) + \ell \cdot \sum_{i=1}^{m} \frac{d_i}{t_i} \cdot \phi\left({}^{e_i}/_{d_i}\right) \leq \tilde{H}^{ub}. \tag{4.4}$$

□

The bound in Equation (4.4) takes into account the additional deadline information. When $d_i = t_i$, $\forall \tau_i \in \mathcal{T}$, then the equality is strict, and the bound in Equation (4.4) is precisely equal to $\tilde{H}^{ub}$. On the contrary, if $\exists \tau_i$, s.t. $d_i \neq t_i$, then the bound in Equation (4.4) is tighter than $\tilde{H}^{ub}$. The case of $\exists \tau_i$, s.t. $d_i \neq t_i$ is not examined in more detail in this thesis. However, future work could, and should, explore the properties of such tasksets further.

Cardinality Bound

We will now show that a given schedule set $\mathcal{K} \subseteq \mathcal{S}$ can only achieve an upper-approximated entropy of $\tilde{H}^{ub}$ if it has cardinality of at least

$$\frac{\ell}{\gcd(e_i/t_i \cdot \ell)}.$$

This bound is important, since it shows that if we want to achieve the upper bound on the upper-approximated entropy from Theorem 1, we must use a schedule set of at least the size given above.

THEOREM 2
Given any schedule set $\mathcal{K} \subseteq \mathcal{S}$, if

$$|\mathcal{K}| < \frac{\ell}{\gcd\left(e_i/t_i \cdot \ell\right)},$$

then the inequality in Equation (4.1) is strict. □

Proof To achieve the upper bound $\tilde{H}^{ub}$, the contribution to the upper-approximated entropy of each slot should be maximized. This happens when the slot entropy H_j is equal to $H_j = \sum_{i=0}^{m} \phi(e_i/t_i)$ for each slot, i.e. each task has a chance to occur equal to its relative frequency (Theorem 1). The contribution of each task to the slot entropy should then be related to the task utilization e_i/t_i. To find a lower bound for k, we can then look at a single slot j. The problem of finding $|\mathcal{K}^*| = k^*$ can then be formulated as an optimization problem.

$$\begin{aligned} \underset{\substack{k \in \mathbb{Z}^+ \\ C_{j,i,\mathcal{K}} \in \mathbb{Z}^+}}{\text{Minimize:}} \quad & k \\ \text{Subject to:} \quad & C_{j,i,\mathcal{K}} = \frac{e_i}{t_i} \cdot k, \quad \forall i \in \{0, \ldots, m\} \end{aligned} \tag{4.5}$$

This means that we have a positive integer number of schedules in the set $\mathcal{K}$ which allows $C_{j,i,\mathcal{K}}$ (the number of times task i appears in slot j in set $\mathcal{K}$) to be a positive integer number for each task τ_i. We perform a variable substitution and define $y = \ell/k$. Minimizing k now becomes equivalent to maximizing y. Temporarily relaxing the requirement that k be an integer, the problem in Equation (4.5) is reformulated as

$$\begin{aligned} \underset{C_{j,i,\mathcal{K}} \in \mathbb{Z}^+}{\text{Maximize:}} \quad & y \\ \text{Subject to:} \quad & C_{j,i,\mathcal{K}} = \frac{e_i}{t_i} \cdot \frac{\ell}{y}, \quad \forall i \in \{0, \ldots, m\} \end{aligned} \tag{4.6}$$

The solution of the problem in Equation (4.6) is that y should be equal to the greatest common divisor of the utilizations multiplied by the hyper-period, $y = \gcd(e_i/t_i \cdot \ell)$, which yields to

$$k^* = \frac{\ell}{\gcd\left(e_i/t_i \cdot \ell\right)}. \tag{4.7}$$

For this to be the solution of the optimization problem in Equation (4.5), k^* must be an integer number. It is known that the utilizations of the taskset (including the idle task) sum to one, $\sum_{i=0}^{m} e_i/t_i = 1$, the constraint for the idle task in the optimization problem of Equation (4.6) can also be written as

$$C_{j,0,\mathcal{K}} = (1 - U) \cdot \frac{\ell}{y} = \left(\frac{\ell}{y} - \frac{\ell}{y} \cdot U\right).$$

The solution of problem (4.6) ensures that $\frac{\ell}{y} \cdot \frac{e_1}{t_1} + \ldots + \frac{\ell}{y} \cdot \frac{e_m}{t_m} = \frac{\ell}{y} \cdot U \in \mathbb{Z}^+$ due to the m constraints for the non idle tasks in the taskset. The value of ℓ/y (equal to k^*) must then be a positive integer number, since $\frac{\ell}{y} \cdot U$ and $C_{j,0,\mathcal{K}}$ are positive integers. This implies that the solution of the problem in Equation (4.6) is also a solution of the problem in Equation (4.5) and $k^* \in \mathbb{Z}^+$. □

Corollary 4
A simple modification of the proof of Theorem 2 shows that for any $\mathcal{K} \subseteq \mathcal{S}$, if

$$|\mathcal{K}| \bmod \frac{\ell}{\gcd(e_i/t_i \cdot \ell)} \neq 0,$$

then $\tilde{H}(\mathcal{K}) < \tilde{H}^{ub}$. This means that the cardinality of $\mathcal{K}$ must be a multiple of $\frac{\ell}{\gcd(e_i/t_i \cdot \ell)}$ in order to achieve the bound in Theorem 2. □

4.2 Example

Applying Theorems 1 and 2 to the example presented in Section 3.2 gives us:

$$\tilde{H}^{ub} = \ell \cdot \sum_{i=0}^{m} \phi\left(e_i/t_i\right) = 4 \cdot \{\phi(1/2) + \phi(1/4) + \phi(1/4)\} = 6,$$

and

$$k^* = \frac{\ell}{\gcd\left(e_i/t_i \cdot \ell\right)} = \frac{4}{\gcd\left(4/2, 4/4, 4/4\right)} = 4.$$

5

Schedule Set Generation for Implicit Deadline Tasksets

This chapter presents an algorithm which aims to generate a solution $\mathcal{K}^*$ to Problem 1 for a taskset $\mathcal{T}$ where tasks have *implicit deadlines*, i.e., for a taskset where $\forall \tau_i \in \mathcal{T}, d_i = t_i$. With constraints corresponding to the fundamental limits proposed in Chapters 3 and 4, a minimum-size solution maximizing the upper-approximated entropy is found, using a Constraint Programming solver.

Pseudocode

Algorithm 1: The `generate_schedules` function which generates an optimal schedule set for a particular task set.

```
1  Function generate_schedules(Taskset ts)
2      solver = ConstraintSolver()
3      grid = initialize_grid(ts)
4      for row ← 0 to nbr_schedules do
5          foreach task in ts do
6              solver.add(ScheduleConstraint(grid, row, task))
7          end
8      end
9      for col ← 0 to hyperperiod do
10         foreach task in ts do
11             solver.add(SlotConstraint(grid, col, task))
12         end
13     end
14     solver.add(FixConstraint(grid, ts))
15     solution = solver.Solve()
16     return solution
```

5.1 Notation

- The **optimal schedule set** is defined as a schedule set ($\mathcal{K}^*$ as described in Problem 1) which reaches the upper bound on the upper-approximated entropy using as few schedules as possible.

- The **optimal number of occurrences** of a task τ_i for a slot j is defined as $n_i^* = k \cdot {}^{e_i}/_{t_i}$ (as stated in Equation (4.5)).

5.2 Fundamental Algorithm Idea

The core of Algorithm 1 relies on a *Constraint Optimization* (also called *Constraint Programming* or *CP* for short) solver. Unlike "ordinary" programming, CP does not specify how or in which order a sequence of instructions should be executed rather than specifying constraints on a solution. One can think of it as the following mathematical problem

$$\begin{aligned} \text{Maximize:} \quad & f(x) \\ \text{Subject to:} \quad & g(x) \leq 0 \\ & h(x) = 0. \end{aligned}$$

A constraint programming problem is very similar to a mathematical optimization problem. Constraints in the form of logical constraints, e.g. "True" or "False" statements, can be seen as equality constraints in mathematics, $h(x) = 0$ above. The inequality constraints in mathematics, $g(x) \leq 0$ above, can be represented in CP using linear constraints, e.g. "sum of x < 5" statements.

Constraint programming defines the domain in which a solution is located, rather than finding an algorithmic solution to the problem. The idea behind Algorithm 1 is then to properly characterize the domain of the solution, such that we will find an optimal schedule set, $\mathcal{K}^*$. A grid of size $k \times \ell$, where $k = k^*$ and ℓ is the hyperperiod of the taskset, is created. This grid will be the foundation upon which we add our constraints.

As presented in Equation (4.5) we need $n_i^* = k \cdot {}^{e_i}/_{t_i}$ number of each task τ_i occurring in each slot to achieve optimality. This is introduced as a constraint. Another constraint introduced is the fundamental limitation of each task $\tau_i = \{e_i, t_i, d_i\}$ having to appear e_i times for each period t_i to ensure that the schedule is valid.

5.3 In Depth

Initialization

Looking at Algorithm 1 the first thing that happens is that we create our solver. The solver used for this thesis is the `constraint_solver` from Google's Operations Research tools (abbreviated *OR-tools*) [Google Optimization Tools, 2018]. The solver will keep track of the constraints on each grid point as well as solve the constrained system when the solution is sought.

A grid of size $k \times \ell$ is created in the form of a dictionary. Each key is the position (i, j) in the grid, in the form of a tuple, and each key's value corresponds to the tasks that could exist in this position (the domain). This could be thought of as a $k \times \ell$ matrix where each element (i, j) corresponds to a schedule i and a slot j and the value is the variable domain.

Example

For the taskset, $\mathcal{T}$, from Section 3.2 we already know that $k^* = \ell = 4$. The grid is therefore initialized as a dictionary with 16 elements $(4 \cdot 4)$, each domain containing all possible tasks (τ_0, τ_1 and τ_2)

$$\left\{ \begin{array}{c} (0,0) : [0,1,2] \\ (0,1) : [0,1,2] \\ \cdots \\ (3,2) : [0,1,2] \\ (3,3) : [0,1,2] \end{array} \right\}.$$

The grid could also be visualized as a matrix. If every grid point in the dictionary was represented with a corresponding matrix position, the matrix would look like

$$\begin{pmatrix} [0,\ 1,\ 2] & [0,\ 1,\ 2] & [0,\ 1,\ 2] & [0,\ 1,\ 2] \\ [0,\ 1,\ 2] & [0,\ 1,\ 2] & [0,\ 1,\ 2] & [0,\ 1,\ 2] \\ [0,\ 1,\ 2] & [0,\ 1,\ 2] & [0,\ 1,\ 2] & [0,\ 1,\ 2] \\ [0,\ 1,\ 2] & [0,\ 1,\ 2] & [0,\ 1,\ 2] & [0,\ 1,\ 2] \end{pmatrix}.$$

Schedule Constraints

After the initialization of the solver and the grid, we would like to start adding constraints to the solver. It is crucial that the constraints are well thought through and that they result in an optimal solution. Fortunately for us, the necessary constraints are logical and do not warrant any explanation aside from the arguments provided in Chapters 3 and 4.

The fundamental constraint that each task $\tau_i = \{e_i,\ t_i,\ d_i\}$ has to execute e_i times each period t_i is imposed on us by the schedulability requirement from Chapter 3. Hence, for each schedule we extract the grid points corresponding

to that schedule. Then, for each task τ_i, we add a constraint that it has to occur e_i number times for each period t_i of those grid points. These constraints will force the solver to adjust to the schedulability criteria.

Example (*contd.*)

For each schedule we want to impose constraints. We represent schedule 0 with the grid points corresponding to this schedule

$$\left\{ \begin{array}{l} (0,0) : [0,1,2] \\ (0,1) : [0,1,2] \\ (0,2) : [0,1,2] \\ (0,3) : [0,1,2] \end{array} \right\}.$$

We know that task $\tau_1 = \{e_1, t_1, d_1\} = \{1, 2, 2\}$ has to execute once every two time slots. This is introduced into the solver by the following constraints

$$\{(0,0) : [0, 1, 2], (0,1) : [0, 1, 2]\} \leftarrow \tau_1 \text{ exist once.}$$
$$\{(0,2) : [0, 1, 2], (0,3) : [0, 1, 2]\} \leftarrow \tau_1 \text{ exist once.}$$

The constraint is applied for task τ_0 and τ_2 as well, except in these cases for the entire schedule (since $t_0 = t_2 = 4 = \ell$).

Slot Constraints

The next constraint we introduce comes from the fact that we need the optimal number of occurrences n_i^* for each task τ_i in each slot to achieve the upper bound on the upper-approximated entropy, $\tilde{H}^{ub}$ (see Theorem 2 with corresponding proof). This is done in a similar fasion to the schedule constraints: by fixating the occurrence rates of the tasks over the grid points in a slot. With this constraint we are sure to achieve an optimal slot entropy, see Equation (4.5).

Example (*contd.*)

Once again we extract grid points. This time the ones corresponding to a slot, e.g. slot 0

$$\left\{ \begin{array}{l} (0,0) : [0,1,2] \\ (1,0) : [0,1,2] \\ (2,0) : [0,1,2] \\ (3,0) : [0,1,2] \end{array} \right\}.$$

We know that $(n_0^*, n_1^*, n_2^*) = (1, 2, 1)$ which implies that

$$\left\{ \begin{array}{l} (0,0) : [0,1,2] \\ (1,0) : [0,1,2] \\ (2,0) : [0,1,2] \\ (3,0) : [0,1,2] \end{array} \right\} \quad \begin{array}{l} \leftarrow \tau_0 \text{ exist once,} \\ \leftarrow \tau_1 \text{ exist twice,} \\ \leftarrow \tau_2 \text{ exist once.} \end{array}$$

This constraint is repeated for each slot in the hyperperiod.

Fix Constraints

The final constraint we introduce is not necessary for optimality, but helps the solver prune the search tree such that the search algorithm will find a solution faster.

We force a task τ_i into the grid such that it achieves its optimal number of occurrences, n_i^*, for each slot. By doing this the search tree gets one less task to consider. We approach this insertion as a boolean constraint where the task with the smallest value of n_i^* is considered the *initial task* and is constrained to certain positions in the grid, such that it will exist n_i^* times in each slot and still meet its deadline. In case of a tie where $n_i^* = n_j^*$, $i < j$ we pick n_i^*. We are choosing the least occurring task as the initial task since it will have the least amount of positions it can be placed in.

Example (*contd.*)

We impose the constraint that the task with the smallest n_i^* should be constrained to certain positions. For $\mathcal{T}$ we see that $(n_0^*, n_1^*, n_2^*) = (1, 2, 1)$. Since $n_0^* = n_2^*$, we constrain task τ_0 such that it has $n_0^* = 1$ number of occurrences in each slot and exists $e_0 = 1$ times over each schedule. This is easiest done by inserting τ_0 into all the diagonal elements

$$\left\{\begin{array}{c} (0,0) : [0,1,2] \\ (0,1) : [0,1,2] \\ \cdots \\ (2,3) : [0,1,2] \\ (3,3) : [0,1,2] \end{array}\right\} \quad \begin{array}{l} \leftarrow (0,0) = 0, \\ \leftarrow (1,1) = 0, \\ \leftarrow (2,2) = 0, \\ \leftarrow (3,3) = 0. \end{array}$$

Search and Solve

The necessary constraints are now added to the solver and the last thing to be done is to let the solver find a solution. The solver creates a binary search tree (also called binary decision tree) and uses the tree to search for a solution. As a root node, the empty grid is choosen.

The search algorithm starts off at the root node by pruning the search tree with regards to the fixed constraints. Thereafter, we choose to use either a possible value, v_i, or not to use that value for the first empty position in the grid. In each node it makes the same distinction: either choose to use a value, v_i, or not using that value. After each decision, the branches are pruned such that the algorithm does not explore invalid paths. Once a solution that satisfies our constraints is found, the algorithm stops and returns it as our solution. The value v_i is choosen randomly from the domain in each node.

If something goes wrong, another path in the binary search tree is choosen. This way we traverse the search tree until a solution is found.

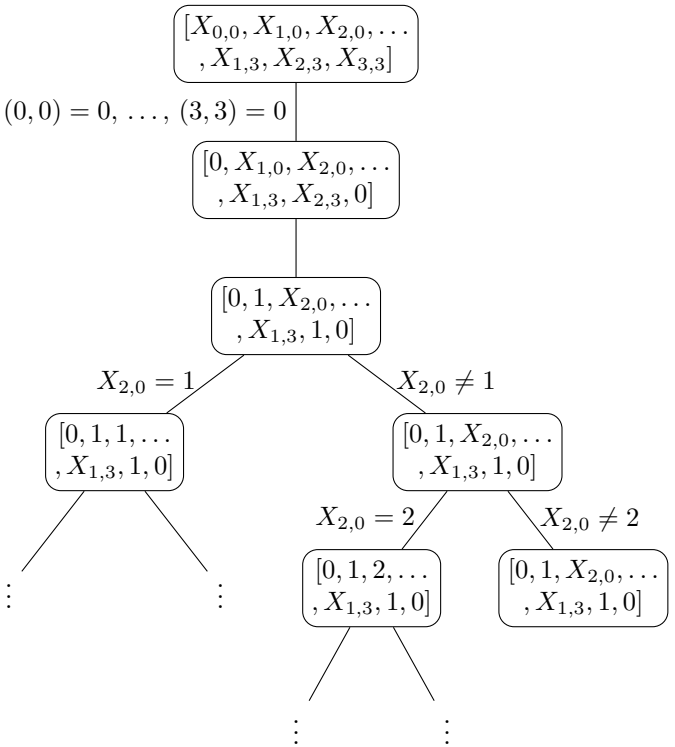


Figure 5.1 A visualization of the search tree the solver uses to search for a solution to the example in Section 3.2.

Example (*contd.*)

To demonstrate the idea behind the solver, denote the unassigned grid points (i, j) with the variable names $X_{i,j}$. This way the grid can be written as an array $\boldsymbol{X} = [X_{0,0}, X_{1,0}, \ldots, X_{2,3}, X_{3,3}]$. The root node is initialized to contain the empty grid $\boldsymbol{X}$. As we have a constraint claiming that $X_{0,0} = X_{1,1} = X_{2,2} = X_{3,3} = 0$ we insert these numbers and prune the search tree. Since $X_{1,1} = 0$ and we have a constraint that τ_1 has to execute once every period of two time slots, $X_{1,0}$ has to contain 1 (as well as for e.g. $X_{2,3}$).

We then have to make a decision: Do we want to assign $X_{2,0} = 1$ or not (1 is here randomly selected from the domain [1, 2])? If we do, we traverse the left branch in Figure 5.1, otherwise we traverse the right branch.

After each decision, the search tree is pruned based on our decision. If we choose $X_{2,0} = 1$ we can neither add task τ_1 to the remaining elements of this slot, since we have constrained τ_1 to appear twice every slot, nor can we add it to $X_{2,1}$, since we constrained τ_1 to appear once every two grid points

in a schedule. This allows us to prune τ_1 from all the concerned grid points, reducing the number of paths in the search tree.

Assume we choose $X_{2,0} \neq 1$, we then have to choose another value instead. The only task left to insert is 2 so if we do not choose this value for $X_{2,0}$ we have no more branches to traverse and an optimal solution to this taskset can not be found. Hence, the rightmost node has got no children. Important to note is that this can never happen for our problem (since we will have found an optimal solution long before we have traversed the entire tree) but the search algorithm works this way.

This tree traversal and pruning algorithm is what the solver does in its final step. There could be, and probably is, many solutions to the constraint programming problem (as stated in Section 3.2). However, we are only interested in one of them. Therefore, as soon as we find one solution, it is returned and the tree traversal is stopped.

6

Experimental Results

This chapter presents a summary of our experimental results. We apply our algorithm to both synthetic tasksets and an industrial case study, validating our approach and measuring its performance.

6.1 Synthetic Tasksets

We tested the algorithm presented in Chapter 5 with synthetic tasksets, composed of 2, 3, 4, and 5 tasks (not including the idle task). For the taskset generation, we used the UUniFast algorithm [Bini and Buttazzo, 2005].

We divided the utilization U into ten groups. For each group $g = [0, 1, \dots 9]$, we generated 100 random numbers $U_i \in [0.02 + 0.1 \cdot g, 0.08 + 0.1 \cdot g]$, to cover a wide spectrum of utilization values. A common maximum hyperperiod length $\ell_{max} = 100$ was specified, such that all synthetic tasksets $\mathcal{T}_i$ had shorter or equal hyperperiod length to the specified one, $\ell_i \leq \ell_{max}$. Each task $\tau_j = \{e_j, t_j, d_j\} \in \mathcal{T}_i$ was assigned a randomized portion (u_j) of the utilization U_i, based on the UUniFast algorithm [Bini and Buttazzo, 2005]. Thereafter, the periods t_j for each task $\tau_j \in \mathcal{T}_i$ were randomized such that $\ell_i \leq \ell_{max}$. Since implicit deadlines were used, we assigned $d_j = t_j$. Finally, the execution times were constructed from the already randomized periods t_j and task utilizations u_j as

$$e_j = \max\left(1,\, \lfloor u_j \cdot t_j \rfloor\right).$$

If $\mathcal{T}_i$ did not achieve an utilization $U = \sum_j {}^{e_j}/_{t_j}$ sufficiently close to U_i, the task utilizations u_j and the task periods t_j were recalculated until U was sufficiently close to U_i. Sufficiently close in this context means within two bounds, $0.02 + 0.1 \cdot g \leq U \leq 0.08 + 0.1 \cdot g$.

We computed the upper-approximated entropy $\tilde{H}^{ub}$, for the taskset according to Theorem 1. We then computed the average contribution of each slot $\tilde{H}^{ub}/\ell$, to be able to compare taskset with different hyperperiods. Finally, we ran our algorithm to generate the schedule set $\mathcal{K}$, with the lowest cardinality

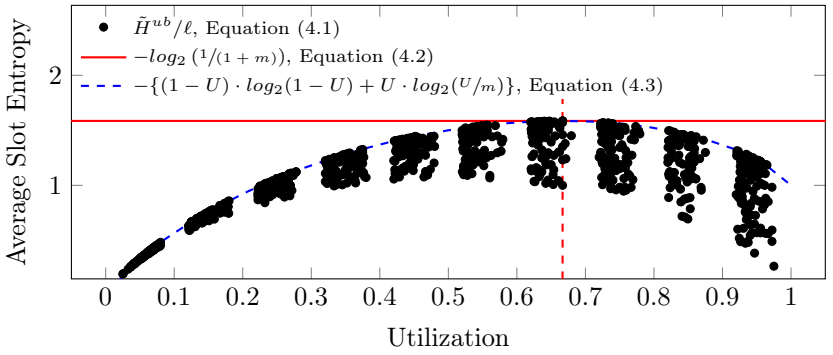


Figure 6.1 Average Slot Entropy of random sets composed of 2 tasks.

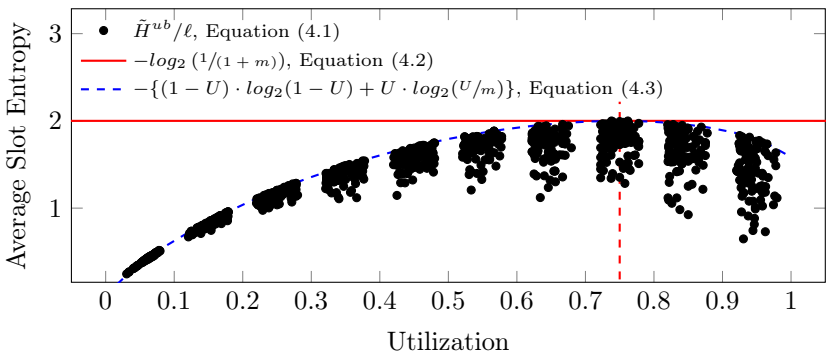


Figure 6.2 Average Slot Entropy of random sets composed of 3 tasks.

k^* given by Theorem 2. The test was executed on an Intel Core i7-3520M CPU @ 2.90GHz.

In principle, the algorithm could run for a long time to find the maximum, so we limited the duration the algorithm was allowed to run with a budget of one minute to compute the schedule set. We found that for all the sets with 2 and 3 tasks, the algorithm was able to return an optimal schedule set within the given time. For the tasksets composed of 4 tasks, only in 7 cases (out of 1000 attempts) the algorithm was not able to find an optimal schedule set within the one minute limit. For the tasksets composed of 5 tasks, the number of failed attempts was 23 out of 1000.

Figures 6.1, 6.2, 6.3, and 6.4 show the average contribution of each slot to the upper-approximated entropy $\tilde{H}^{ub}/\ell$, respectively in the case of tasksets composed of 2, 3, 4, and 5 tasks. The dots in the figure show the maximum upper-approximated slot entropy, the constant, red lines represent the (non-

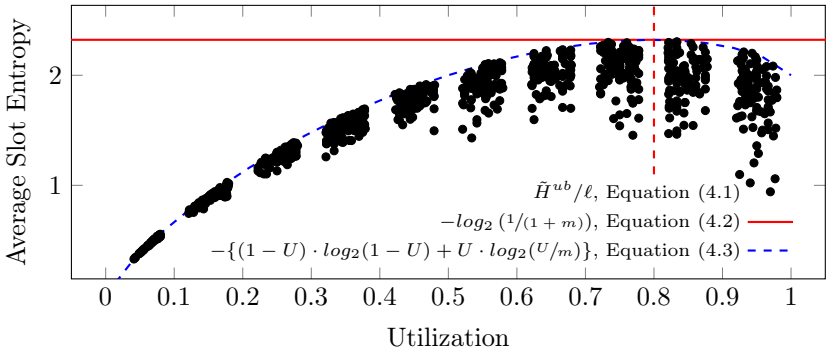


Figure 6.3 Average Slot Entropy of random sets composed of 4 tasks.

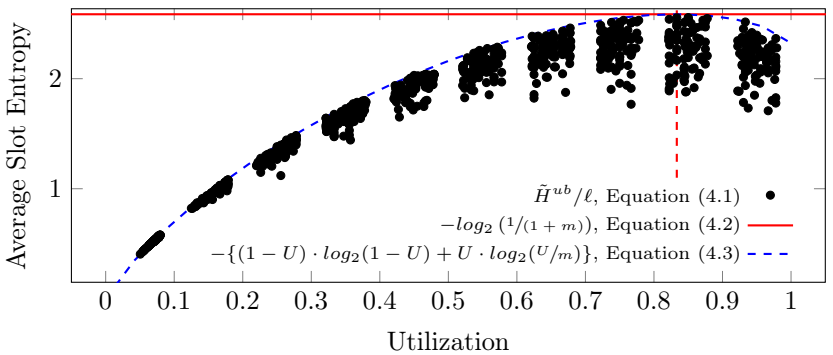


Figure 6.4 Average Slot Entropy of random sets composed of 5 tasks.

tight) bound introduced by Corollary 1, while the dashed line represents the (not tight) bound discussed in Corollary 2, Equation (4.3). As can be seen, for some of the tasksets, the constraints imposed by the execution times and the periods allow the upper-approximated entropy to reach this bound, but in other cases the bound presented in Theorem 1 is tighter. This is due to the fact that we could have a skewed distribution of the taskset utilization U_i over the individual tasks. For example, given a taskset composed of 2 tasks and total utilization $U_i = 0.5$ distributed such that task τ_1 has a utilization of 0.45 and task τ_2 has a utilization of 0.05. The upper-approximated entropy will then be relatively low since τ_2 has a low occurence rate.

One can also notice that when the utilization increases, the variance in the achieved upper-approximated entropy increases. This is because a higher utilization introduces tighter constraints on the achievable upper-approximated entropy. Another important result, taken from Figures 6.1, 6.2, 6.3, and 6.4,

is the fact that the upper-approximated entropy reaches the maximum of the bound discussed in Corollary 2 for $^{m}/_{(m+1)}$ – marked with the dashed, red, vertical line – m being the number of tasks. Consequently, it is not optimal (from a randomization point of view) to maximize the utilization of your taskset. Having the taskset utilization $U = \sum_{i=1}^{m} {}^{e_i}/_{t_i} = {}^{m}/_{(m+1)}$, would benefit the randomization. This might be a very important feature in how we try to balance our taskset utilization in future schedules.

Finally, In most cases, the bound given by Corollary 1 is unreachable, due to the constraints introduced by the tasks characteristics. In fact, it is only reachable for when the utilization of the taskset is $^{m}/_{(m+1)}$. We know the bound from Corollary 1, $log_2(m+1)$, and we know that the highest upper-approximated entropy we could reach occurs when $U = {}^{m}/_{(m+1)}$, or in other words when all tasks could have equal probability of appearing. Given a taskset with m tasks where each task has a relative frequency of $u_j = {}^{1}/_{(m+1)}$, the bound from Equation (4.2) in Corollary 1 coincides with the bound from Equation (4.3) in Corollary 2

$$\begin{aligned}
\frac{\tilde{H}^{cor1}}{\ell} &= -(1-U)\cdot log_2(1-U) - U\cdot log_2\left(\frac{U}{m}\right) = \left[U = \frac{m}{m+1}\right] = \\
&= -\left(\frac{1}{m+1}\right)\cdot log_2\left(\frac{1}{m+1}\right) - \frac{m}{m+1}\cdot log_2\left(\frac{1}{m+1}\right) = \\
&= \frac{1}{m+1}\cdot\left(log_2(m+1) + m\cdot log_2(m+1)\right) = \\
&= log_2(m+1) = \frac{\tilde{H}^{cor2}}{\ell}.
\end{aligned}$$

6.2 Case Study

We have also tested our algorithm with a real-world case study. For this, we have used the Research Open-Source Avionics and Control Engineering (ROSACE) case study [Pagetti et al., 2014]. ROSACE is a multi-periodic extension of the mono-periodic longitudinal controller presented in [Gervais et al., 2012]. ROSACE implements a simple, but representative, longitudinal flight controller. The controller is designed to ensure that the aircraft maintains average cruise conditions for the flight, in terms of altitude and speed. We have extracted the task parameters for the case study.

The taskset is composed of eight independent, periodic tasks, $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_8\}$. The case study provides a specification of the taskset parameters in terms of periods and execution times. The deadline of each task is assumed to be equal to its period, making this a good case study for our implicit deadline taskset algorithm. The eight tasks of the controller have

	e_i	t_i	d_i
τ_1	1	100	100
τ_2	1	100	100
τ_3	1	100	100
τ_4	1	100	100
τ_5	1	100	100
τ_6	1	200	200
τ_7	1	200	200
τ_8	1	200	200

Table 6.1 Periods and execution times (expressed in time units, where one time unit corresponds to 0.1ms) for the taskset of the ROSACE case study

activation frequencies that belong to two groups: 50Hz and 100Hz, corresponding to periods of 20ms and 10ms respectively. In the following we assume that a time unit for our scheduler is 0.1ms, i.e., we generate schedules where each slot corresponds to a time interval of 0.1ms. With this assumption, the parameters of the ROSACE case study are given in Table 6.1.

With the given parameters from Table 6.1, the hyper-period ℓ of the taskset corresponds to 200 time units, $\ell = 200$. The maximum achievable upper-approximated entropy is computed using Equation (4.1) (according to Theorem 1) and results in $\tilde{H}^{ub} = 107.502$. Using Theorem 2 (Equation (4.7)) we can compute the minimum number of schedules needed to obtain a schedule set reaching the mentioned upper bound. The result $k^* = 200$ was obtained.

Our algorithm is able to find the optimal solution in 3.44 seconds on an Intel Core i5-3337U CPU @ 1.80GHz, demonstrating that our approach is viable to use also in industrially relevant case studies.

7

Conclusion and Future Work

This chapter concludes the thesis. We summarize the results obtained and highlight future research directions.

7.1 Summary

Considering the rapid increase in real-time embedded applications, the growing threat from side-channel attacks exploiting system information to disrupt or jam the systems must be impeded. With upper-approximated entropy measuring the diversity of a schedule set, we introduce schedule set randomization in order to impede temporal inference from surreptitious attackers as well as keep the schedulability. Based on this premise, we have established analytical and mathematical bounds on the upper-approximated entropy of a schedule set and designed an algorithm to create a schedule set that reaches these bounds. With the analytically acquired results we aim to provide a foundation for future research within the field of embedded security systems.

7.2 Future Work

Security is an important concern for modern real-time and embedded systems and recent research has only begun to scratch the surface. It is imperative that more focus is put on this area of security, given the prominence and importance of embedded devices in many aspects of life, from the Internet of Things to cell phones and cars. This has generated an increase in targeted attacks towards such systems.

This thesis has provided analytical results on how to optimally mitigate side-channel attack. An example of how this work could be combined with other security measures is provided here. *Message Authentication Codes*

(MACs) are used to authenticate that the messages acquired by an embedded system are valid and not falsified by a corrupted sensor or an attacker, checking whether the message has been tampered with. However, MACs are immensely time consuming and are therefore often avoided in embedded systems with hard real-time constraints. A solution to this problem has been suggested by the authors of [Lesi et al., 2017]. They propose a method that instead of authenticating every message to an embedded system (often making schedulability infeasible) only authenticates some tasks, ensuring schedulability in each hyperperiod. Given the schedule set randomization method presented in this thesis, the idle task is used to further increase the upper-approximated entropy, but other than that it does not affect our system. Using the idle task in the schedule set randomization method as a time slot where the constrained MACs (as described in [Lesi et al., 2017]) can execute combines two different security measures. This approach might impede attackers from using corrupted sensor signals as well as side-channels to disrupt the system.

Based on the results of this thesis, there is still a lot to explore. This report barely touched the subject of what happens when the deadlines are not implicit, or in other words when $d_i \neq t_i$. It is a generalization of what has been presented in this thesis and it would be interesting to see how the theory presented here could be applied to the general case. The assumption that the optimal probability that a task τ_i appears in each slot j is equal to e_i/t_i is no longer valid and would have to be revisited. If similar bounds to the ones presented in Chapter 4 could be found for the non-implicit deadline case, the number of tasksets possible to obfuscate would be significantly increased. This would also open up for research regarding randomization of tasksets including task latency, assumptions regarding arrival time, jitter, etc.

In reality, there could be a hard constraint that a task in some cases has to access the CPU at a given precise moment. In that case, we could for example combine schedules, from the feasible set (especially for tasksets with harmonic periods). By merging two feasible schedules, and mixing part of another schedule that satisfies the hard constraint of a task accessing the CPU at precise time instants, we might preserve the properties that we have obtained in terms of upper-approximated scheduling entropy with a modification of the proposed technique. Context switches and their costs were also not considered in this work, as well as the presence of interrupts and interrupt service routines.

In this thesis – as well as in previous work [Yoon et al., 2016; Kruger et al., 2018] – we have assumed that the vulnerability to side-channel attacks is reduced when schedule randomization is employed. This is in most cases a reasonable assumption. However, in control applications, e.g. embedded controllers, the control performance is reduced when jitter (deviation from the true period) is introduced [Cervin et al., 2003]. The performance degredation

we acquire from using schedule randomization in an embedded controller might be worth looking further into. Tools for analyzing this type of performance has been developed by the authors of [Cervin et al., 2003], called Jitterbug and TrueTime. Future work might want to utilize these tools to analyze whether or not schedule randomization is a valid method to use in embedded control systems.

Applying the theory presented in this thesis to a multiprocessing setup would also be an interesting research area. To apply the schedule set idea to a multiprocessing environment might increase stochasticity and widen the applicability of our proposal. Scheduling tasks, using the schedule set method presented in this thesis over a fixed number of cores is not an improbable idea, and might prevent attacks on multi-core embedded systems as well.

The proposal developed in this master thesis project might need improvements as well. Even though it works well enough for our benchmark tests, the algorithm might get better with some additional clever constraint to prune the search space even further. The solver itself might also not be optimal. There are a few constraint programming toolboxes on the market that might be worth benchmarking against one another to see which one gives the best results. OR-Tools LCG[1], JaCoP[2], Choco 4[3], and Opturion CPX[4] are some of the toolboxes that have performed the best in the last couple of years and it might be a good idea to test them.

7.3 Conclusion

A minimum-size schedule set maximizing the upper-approximated entropy has been found analytically using only the taskset characteristics. Hence, minimizing side-channel attack vulnerability is possible and has been shown to work. Bounds on the entropy-like notion of upper-approximated entropy as well as on the minimum schedule set size has been provided and discussed. These bounds provide a foundation upon which future security measures against side-channel attacks could be based. Furthermore, given these bounds, a proposal of how to generate an optimal schedule set has also been discussed. Although analytical results have been provided, they have not been tested in an attack simulation. Furthermore, research treating non-implicit deadlines has yet to be conducted. Given the results presented in this thesis, the designers of future real-time systems have acquired additional tools in designing *safety critical* systems.

[1] `https://developers.google.com/optimization/`
[2] `https://osolpro.atlassian.net/wiki/spaces/JACOP/pages/24248331/JaCoP+Download`
[3] `http://www.choco-solver.org/`
[4] `https://www.opturion.com/`

Bibliography

Årzén, K.-E. (2014). "Real-time control systems". In: KFS AB, Lund.

Bini, E. and G. C. Buttazzo (2005). "Measuring the performance of schedulability tests". *Real-Time Systems* **30**:1-2, pp. 129–154. ISSN: 0922-6443. DOI: 10.1007/s11241-005-0507-9.

Bos, J. W., C. Hubain, W. Michiels, and P. Teuwen (2016). "Differential computation analysis: hiding your white-box designs is not enough". In: Gierlichs, B. et al. (Eds.). *Cryptographic Hardware and Embedded Systems – CHES 2016*. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 215–236. ISBN: 978-3-662-53140-2.

Cervin, A., D. Henriksson, B. Lincoln, J. Eker, and K.-E. Årzén (2003). "How does control timing affect performance? analysis and simulation of timing using jitterbug and truetime". *IEEE Control Systems* **23**:3, pp. 16–30. ISSN: 1066-033X. DOI: 10.1109/MCS.2003.1200240.

Chen, C., A. Ghassami, S. Mohan, N. Kiyavash, R. B. Bobba, R. Pellizzoni, and M. Yoon (2017). "A reconnaissance attack mechanism for fixed-priority real-time systems". In: vol. abs/1705.02561. arXiv: 1705.02561. URL: http://arxiv.org/abs/1705.02561.

Gervais, C., J.-B. Chaudron, P. Siron, R. Leconte, and D. Saussié (2012). "Real-time distributed aircraft simulation through hla". In: *16th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*. DS-RT.

Google Optimization Tools (2018). *The cp solver*. URL: https://developers.google.com/optimization/cp/cp_solver (visited on 03/28/2018).

Jiang, K., L. Batina, P. Eles, and Z. Peng (2014). "Robustness analysis of real-time scheduling against differential power analysis attacks". In: *2014 IEEE Computer Society Annual Symposium on VLSI*, pp. 450–455. DOI: 10.1109/ISVLSI.2014.11.

Kocher, P., J. Jaffe, and B. Jun (1999). "Differential power analysis". In: Wiener, M. (Ed.). *Advances in Cryptology — CRYPTO' 99*. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 388–397. ISBN: 978-3-540-48405-9.

Kruger, K., M. Völp, and G. Fohler (2018). "Vulnerability analysis and mitigation of directed timing inference based attacks on time-triggered systems". In: *Euromicro Conference on Real-Time Systems*. ECRTS. LIPICS.

Lesi, V., I. Jovanov, and M. Pajic (2017). "Security-aware scheduling of embedded control tasks". *ACM Transactions on Embedded Computing Systems* **9**:4.

Mejri, M. N., J. Ben-Othman, and M. Hamdi (2014). "Survey on vanet security challenges and possible cryptographic solutions". *Vehicular Communications* **1**:2, pp. 53 –66. ISSN: 2214-2096. DOI: https://doi.org/10.1016/j.vehcom.2014.05.001. URL: http://www.sciencedirect.com/science/article/pii/S2214209614000187.

Pagetti, C., D. Saussié, R. Gratia, E. Noulard, and P. Siron (2014). "The ROSACE case study: from simulink specification to multi/many-core execution". In: *IEEE Real-Time and Embedded Technology and Applications Symposium*. RTAS.

Pietro, R. D., S. Guarino, N. Verde, and J. Domingo-Ferrer (2014). "Security in wireless ad-hoc networks – a survey". *Computer Communications* **51**, pp. 1 –20. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2014.06.003. URL: http://www.sciencedirect.com/science/article/pii/S0140366414002242.

Shannon, C. E. (1948). "A mathematical theory of communication". In: vol. 27. 3, pp. 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

Spreitzer, R., V. Moonsamy, T. Korak, and S. Mangard (2017). "Systematic Classification of Side-Channel Attacks: A case study for mobile devices". In: *IEEE Communications Surveys and Tutorials*. RTAS, pp. 1–24.

Yoon, M., S. Mohan, C. Chen, and L. Sha (2016). "TaskShuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems". In: *IEEE Real-Time and Embedded Technology and Applications Symposium*. RTAS, pp. 111–122.

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER'S THESIS

Date of issue
June 2018

Document Number
TFRT-6059

Author(s)
Nils Vreman

Supervisor
Martin Maggio, Dept. of Automatic Control, Lund University, Sweden
Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden (examiner)

Title and subtitle
Minimizing Side-Channel Attack Vulnerability via Schedule Randomization

Abstract
Predictable and repeatable execution is the key to ensuring functional correctness for real-time systems. Scheduling algorithms are designed to generate schedules that repeat after a certain amount of time has passed. However, this repeatability is also a vulnerability when side-channel attacks are considered.

Side-channel attacks are attacks based on information gained from the implementation of a system, rather than on weaknesses in the algorithm. Side-channel attacks have exploited the predictability of real-time systems to disrupt their correct behavior.

Schedule Randomization has been proposed as a way to mitigate this problem. Online, the scheduler selects a schedule among a set of available ones, trying to achieve an execution trace that is as different as possible from previous ones, therefore minimizing the amount of information that the attacker can gather.

This thesis investigates fundamental limitations of schedule randomization for a generic taskset. We then propose an algorithm to construct a set of schedules that achieves a differentation level as high as possible, using the fewest number of schedules, for tasksets with implicit deadlines. The approach is validated with synthetically generated tasksets and the taskset of an industrial case study, showing promising results.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-42

Recipient's notes

Security classification