Information Leakage as a Scheduling Resource

Fabrizio Biondi, Mounir Chadli, Thomas Given-Wilson, Axel Legay

Information Leakage as a Scheduling Resource

Fabrizio Biondi*, Mounir Chadli*, Thomas Given-Wilson*, and Axel Legay*
*Inria

Email: {fabrizio.biondi,thomas.given-wilson,axel.legay}@inria.fr, mounir.chadli@irisa.fr

***Abstract*—High-security processes typically have to load confidential information, such as encryption keys or private data, into memory as part of their operation. In systems with a single shared memory, when high-security processes are switched out due to context switching, confidential information may remain in memory and be accessible to low-security processes. This paper considers this problem from the perspective of scheduling. A formal model supporting preemption is introduced that allows: reasoning about leakage between high- and low-security processes, and producing information-leakage aware schedulers. Several information-leakage aware heuristics are presented in the form of compositional pre- and postprocessors as part of a more general scheduling approach. The effectiveness of such heuristics is evaluated experimentally, showing them to achieve significantly better schedulability than the state of the art.**

I. Introduction

This paper considers a shared memory system where processes are classified as either *high-security* or *low-security*. High-security processes work with confidential information that should not be accessed by low-security processes. Typically, this includes loading confidential information into memory for use within high-security processes. Examples of such confidential information include encryption keys, medical data, and bank details. This confidential information may be vital to the operation of the high-security processes, but must also be tightly controlled and not be accessed by low-security processes. For instance, in an embedded sensor, high-security encryption processes are handling encryption keys that must not be accessed by low-security data compression processes.

However, high-security processes may not properly flush confidential information from the shared memory, or context switching may interrupt their execution before such flushing can be applied. Consequently, confidential information remaining in the shared memory becomes available to low-security processes.

Consider the small example in Fig. 1, written in Intel x86-64 assembly code for Linux compiled to ELF format[1]. There are two processes: `Process 1` doing some (trivial) encryption operations, and `Process 2` attempting to access the encryption key. `Process 1` takes a *key* `$KEY` and a *message* `$MSG` then encrypts the message with the key using an exclusive or `XOR` operation. The result is then output to the disk (represented by `$DISK1`). `Process 2` writes to a different disk location (represented by `$DISK2`) the content of register `edx`. It is clear that if `Process 2` is executed after the first operation and before the fourth operation of `Process 1`, then the value of the key is leaked.

```
; Process 1:
 mov    r13,$KEY   ; load key to register r13
 mov    r14,$MSG   ; load message to register r14
 xor    r14,r13    ; encrypt the message (r14) with
                   ; the key (r13) using XOR, store
                   ; the result in register r14
 xor    r13,r13    ; wipe value of key (r13) by XOR
                   ; with the key (r13)
 out    $DISK1,r14 ; output the ciphertext (r14)

; Process 2:
 out    $DISK2,r13 ; output register r13 (that may
                   ; store the key) to disk
```

Fig. 1: Example of processes exhibiting schedule-dependent confidential information leakage.

If a scheduler is aware of a process' access level, then the scheduler can take action to prevent confidential information being leaked to low-security processes. Recent work [13], [14] has explored these kinds of problems in a real-time setting by scheduling a complete memory flush after any high-security process that is followed by a low-security process. However, this provides only limited options to the scheduler since such a complete memory flush is expensive and may prevent real-time tasks from meeting their deadlines. Further, when such flushing is not possible, the current approaches do not quantify how significant the information leakage is, simply considering there to have been some leakage.

This paper proposes treating *confidentiality*, measured by the *resulting leakage* of secure information, as a quantitative resource that the scheduler can exploit. This allows for better quantification of the resulting leakage in different scenarios, as well as having a clear measure of the cost of different scheduling choices. Further, this allows for the creation of schedulers that can make better scheduling choices and also respect confidential information leakage constraints.

Consider the standard *workflow model* [3], [9], where a set of *tasks* periodically produce *jobs* that have to be scheduled to complete before their given *deadlines*. The workflow model is here extended by considering tasks to be composed of *steps*, each of which has an *execution time*, *leakage value*, and *security level*. Each one of these steps is implicitly an atomic sequence of actions that can be taken within a task

[1]This example works due to the register `r13` not being set to any specific value at process initialization on X86-64 hardware (https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf) or by the ELF initialization (http://lxr.linux.no/linux+v3.2.4/arch/ia64/include/asm/elf.h).

without preemption by the scheduler. Thus a task consists of an ordered sequence of steps to be performed, that yields the total behavior of the task.

Using this extended workflow model, schedulers can operate upon steps rather than jobs, and so implement preemption while also being able to reason about leakage in a fine-grained manner. This supports offline schedulers in periodic systems that can plan an optimal strategy, as well as online schedulers for arbitrary systems that optimize using the knowledge available. Further, schedulers can be considered that operate over thresholds or within certain security constraints.

The approach in this paper is able to easily capture prior work [13], [14] by inserting a *flush* task that has a known runtime cost and ensures a complete memory wipe (and thus zero resulting leakage). When a high-security job would be followed by a low-security job, a flush is inserted between them. This enforces zero resulting leakage, but often results into poor schedulability due to the high cost of frequently flushing the whole memory.

By considering the flush task to be always available (rather than at prescribed times), schedulers can add flushes when this reduces resulting leakage and still achieve schedulability. Indeed, it is often possible to achieve zero resulting leakage even when flushing after every high-security job is not possible.

More generally, this paper proposes heuristic algorithms to achieve efficient scheduling while reducing resulting leakage, i.e. the amount of confidential information that can be accessed by low-security jobs.

Scheduling algorithms produce a schedule for a set of tasks. Standard scheduling algorithms are extended with a *preprocessing* and a *postprocessing* phase. Preprocessing modifies the set of tasks to be scheduled, while postprocessing modifies the schedule produced by a scheduling algorithm to yield another schedule. Several pre- and postprocessors designed to reduce leakage are introduced in this paper.

Experimental results are presented that demonstrate the trade-off between leakage and schedulability. These results show that this approach schedules, with good (or zero) resulting leakage, sets of tasks that are not schedulable according to the state of the art. Further, different pre- and postprocessors and their impact on the resulting leakage are evaluated.

Key Contributions

The key contributions of this paper are as follows.

- A model to reason quantitatively on the amount of information leaked by scheduling tasks with different security levels on a shared memory system.
- A scheduling approach with compositional and specialized pre- and postprocessors that allows scheduling tasks while reducing the amount of confidential information leaked.
- Several heuristic pre- and postprocessing algorithms that can be exploited to reduce leakage.
- Experimental evaluation of the combinations of the pre- and postprocessors, showing that the approach provides significantly better schedulability and lower information leakage than the state of the art.

The structure of the paper is as follows. Section II recalls background information. Section III extends the workflow model to support preemption and fine-grained leakage. Section IV presents the three-phase approach to scheduling used here, with algorithms for pre- and postprocessing. Section V highlights and discusses the experimental results. Section VI discusses variations and extensions to the model and algorithms. Section VII concludes and considers future work.

II. Background

This section recalls background information useful for understanding the rest of the paper and discusses related work. In particular, this section overviews the workflow model, scheduling algorithms, and information leakage.

A. Workflow Model

This section recalls the workflow model, a standard model for the scheduling of periodic tasks [3], [9]. Section III extends the workflow model to account for the possible leakage of confidential information.

Assume an infinite time divided into discrete time units indexed by natural numbers. Let Γ be a set of independent periodic *tasks* $\{\mathcal{T}_\alpha, \mathcal{T}_\beta, \dots\}$ with each task $\mathcal{T}_x \in \Gamma$ having a *period* P_x, an *execution time* E_x, and a *relative deadline* D_x. A *job* $\tau_{x,k}$ is produced by the activation of a task $\mathcal{T}_x \in \Gamma$ at *release time* $R_{x,k} = (k-1)P_x$, $\forall k \in \mathbb{N}_0$. Each job $\tau_{x,k}$ must be completed before its *absolute deadline* $A_{x,k} = R_{x,k} + D_x$.

The *hyperperiod* H_Γ of a set of tasks Γ corresponds to the least common multiplier of the period P_x of each task $\mathcal{T}_x \in \Gamma$:

$$H_\Gamma = lcm\{P_x \mid \mathcal{T}_x \in \Gamma\} \quad .$$

B. Scheduling Algorithms

This paper uses two standard scheduling algorithms to schedule the jobs produced by sets of tasks: Earliest Deadline First (EDF) and Least Slack First (LSF). Both are simple and widely used offline scheduling algorithms based on dynamic priority of the jobs being scheduled.

EDF determines the priority of jobs according to their absolute deadline. At any given point in time, out of the currently available jobs, the job with the earliest absolute deadline is scheduled first.

LSF determines the priority of jobs according to their amount of *slack*. This slack is calculated for a job $\tau_{x,k}$ according to the formula $A_{x,k} - t - E_x$ where t is the current time. At any given point in time, out of the currently available jobs, the job with the least slack is scheduled first.

C. Related Work

1) Confidentiality of Real-Time Systems: Real-time systems need to communicate with the outside world, such as receiving data from sensors or communicating with other systems, sometimes over unsecured networks. This communication has allowed attacks against even air-gapped industrial control systems [8].

The real-time scheduling requirement itself can be exploited to generate additional vulnerabilities. For instance, a process can modulate its use of a resource to affect the scheduling of another process, and use this to covertly transmit information [15], [16].

Further vulnerabilities can occur in any system with shared memory. When processes with different security levels share the same memory resources, it is possible for low-security processes to access confidential information that should only be accessed by high-security processes [13]. Using separated memory for processes with different security levels is expensive, particularly if the system has more than two security levels. Mohan et al. [13] consider a shared-memory scenario where low-security processes executing after high-security processes could access the high-security processes' memory space resulting in information leakage. To prevent this, they propose completely flushing the memory after the execution of high-security processes when followed by a low-security process. In [14], Pellizzoni et al. generalize this work by introducing a binary relation NO-LEAK on tasks, where NO-LEAK($\mathcal{T}_x, \mathcal{T}_y$) holds if no leakage can occur from $\mathcal{T}_x$ to $\mathcal{T}_y$. The authors also determine the number of memory flushes needed to enforce the NO-LEAK relation, and consequently construct a preemptivity-assignment scheduling algorithm. This work proposes a more fine-grained approach to confidentiality in similar scenarios.

Formal analysis of scheduling system under resource constraints has been performed by Kim et al. [11], [12]. The proposed approach can be extended to confidentiality as a resource using the model proposed in this paper.

2) Information Leakage: Information leakage quantifies the amount of confidential information leaked by a system, and is widely used to measure of the (in)security of the system [1], [2], [4], [10].

In this paper, leakage is used to measure the amount of confidential information that a high-level job leaves in the shared memory at different moments of its execution. The unit of measure of leakage is not relevant since it depends on the specific application. For instance, if the confidential information is a private key, leakage could measure the number of bits of the key that are leaked. Alternatively, leakage could measure the number of confidential packets leaked from a secure transmission. Therefore, the same leakage model can be used with different leakage measures, where zero leakage represents no loss of confidential information.

III. Model

This section introduces the key concepts and model of the system being scheduled. The model here is based upon the workflow model recalled in Section II. The extension here is to represent more precise information about the internal operations and preemptivity of tasks by dividing them into *steps*. These steps include their own execution time (like a task or job from the workflow model), and are extended to include leakage value and security level. Special tasks are also added to model other operations of the system. The rest of this section details this extended model and presents illustrative examples that motivate the choices in this paper.

A. Concept

This section considers concepts and motivations behind the model presented in this paper. In particular, the division of tasks into steps, how to account for leakage in practice, and justification for special tasks.

1) Steps: This model considers the possibility to divide tasks into fine-grained steps. A step represents an atomic sequence of operations that cannot be interrupted by preemption. The practical implementation of steps depends on the architecture and granularity of the scheduling system.

The model is agnostic to step implementation details as long as an execution time, leakage value, and security level can be defined for each step. The most fine-grained approach would be to consider each CPU operation as a step. For instance, `Process 1` in Fig. 1 would be represented as a task divided into five steps. Thus, a task could be preempted after each CPU operation. Although very simple, in practice this approach is too fine-grained. In lightweight and embedded systems it is common to delegate part of the handling of preemption and atomicity to the programmer, so it is reasonable to consider that the programmer themself could define the steps.

2) Special Tasks: This paper considers two special tasks representing special system operations: *flush* and *wait*.

The flush task flushes all confidential information from the shared memory, for instance by overwriting the entire memory with zeroes. This preserves compatibility with the state of the art [13], [14] where flushing is used as the main tool to preserve confidentiality.

The wait task represents idle processor time. Apart from the obvious use, Section III-C2 shows that the scheduling of idle time can impact the confidentiality of the system.

3) Leakage Values: The leakage value of a step represents the amount of confidential information that would be leaked to an attacker able to read the shared memory just after the steps' execution. The model does not constrain the way the leakage value is obtained: leakage can be added by the programmer as an annotation, computed by an automatic tool [5], [6], [7], [17], or possibly both.

For instance, the programmer could specify critical zones in which the program must not be interrupted, and the leakage values would be computed automatically by a tool (for both critical and non-critical zones).

An alternative, variable-based approach would be to have the programmer annotate some variables as containing confidential information at a certain point (and as cleared of confidential information at a later point). Taint analysis can be used to identify which variables are tainted at each point. Information leakage quantification can be used to quantify leakage from the tainted variables.

B. Formal Model

This section formalizes the extended workflow model.

1) Steps, Tasks and Jobs:

Definition Formally, each *step* is a tuple

$$\mathcal{S}(E, L, X)$$

where E denotes the (worst case) *execution time* that the step takes to be completed, L denotes its (potential) *leakage value*, and X denotes its security level (either *high* $\top$ or *low* $\bot$).

The (potential) leakage value L of a step $\mathcal{S}$ is a measure of the amount of confidential information left in memory at the completion of $\mathcal{S}$. Again, the exact meaning of the leakage value is unimportant here.

Here $\top$ indicates that the step contains confidential information and therefore is high-security. Similarly, $\bot$ indicates that the step should not have access to confidential information and therefore is low-security. Since $\top$ and $\bot$ are used to indicate whether the step has access to confidential information, $\bot$ steps typically have leakage zero. This is not a strict requirement of the model and shall be elaborated upon in Section VI-A2.

For instance, consider `Process 1` in Fig. 1. Each assembly instruction can be represented by a single step with an execution time of one time unit and a security level of $\top$. The first three instructions have a leakage value of one, representing the fact that one word of confidential information (the key) is in the shared memory (in register `r13`). However, the remaining instructions have a leakage value of zero since the fourth instruction wipes `r13`.

The system operates with a set of *tasks* $\Gamma = \{\mathcal{T}_\alpha, \mathcal{T}_\beta, \dots\}$.

Definition Each *task* $\mathcal{T}_x \in \Gamma$ is a tuple

$$\mathcal{T}_x(P_x, D_x, \widehat{\mathcal{S}_x})$$

where P_x is the *period* of the task, D_x is its *relative deadline*, and $\widehat{\mathcal{S}_x}$ is a sequence of *steps* $\mathcal{S}_{xa}, \mathcal{S}_{xb}, \dots$ making up the ordered actions of the task.

Tasks are named with Greek letters, e.g. $\mathcal{T}_\beta$. Steps are named with the corresponding task's Greek letter and a Latin letter in alphabetical order, e.g. step $\mathcal{S}_{\beta c}$ represents the third step of task $\mathcal{T}_\beta$.

Observe that `Process 1` in Fig. 1 can be modeled by the following task:

$$\begin{aligned}\mathcal{T}_\alpha \;=\; \mathcal{T}(P_\alpha, D_\alpha, &\mathcal{S}_{\alpha a}(1,1,\top)\\ &\mathcal{S}_{\alpha b}(1,1,\top)\\ &\mathcal{S}_{\alpha c}(1,1,\top)\\ &\mathcal{S}_{\alpha d}(1,0,\top)\\ &\mathcal{S}_{\alpha e}(1,0,\top))\ .\end{aligned}$$

Similarly, `Process 2` in Fig. 1 can be modeled by the following task:

$$\mathcal{T}_\beta \;=\; \mathcal{T}(P_\beta, D_\beta, \mathcal{S}_{\beta a}(1,0,\bot))\ .$$

Definition Each *job* $\tau_{x,k}$ is created by the activation of the task $\mathcal{T}_x$ at *release time* $R_{x,k} = (k-1)P_x$ for $k \in \mathbb{N}_0$, and is a tuple

$$\tau_{x,k}(R_{x,k}, A_{x,k}, \widehat{\mathcal{S}_{x,k}})$$

where $A_{x,k} = R_{x,k} + D_x$ is the job's *absolute deadline*, and $\widehat{\mathcal{S}_{x,k}}$ is the sequence of steps inherited from task $\mathcal{T}_x$.

Jobs are named with the corresponding task's Greek letter and the number k, so job $\tau_{\beta 4}$ is the fourth job generated by task $\mathcal{T}_\beta$ and step $\mathcal{S}_{\beta 4c}$ is the third step of job $\tau_{\beta 4}$.

For simplicity, a task (resp. job) will be referred to as $\top$ or $\bot$ when all steps within that task (resp. job) are either $\top$ or $\bot$, respectively.

2) Flush and Wait: The model requires a task to represent complete flushing of the memory. The *flush* task is defined by

$$\mathcal{T}_\mathcal{F}(-,-,\mathcal{S}_\mathcal{F}(E_\mathcal{F}, 0, \top))$$

where $E_\mathcal{F}$ is the execution time to completely flush of memory. Observe that *after flushing the memory the leakage is reduced to zero*. This is achieved by the single step $\mathcal{S}_\mathcal{F}(E_\mathcal{F}, 0, \top)$ that takes all the execution time of the flush task and has a zero leakage value. Since the flush task is always available to be scheduled, it has no defined period or deadline (denoted here as -), being able to scheduled (or not) at whim. The security level of flush is $\top$ since it is acceptable for flush to have access to confidential information, and for use in calculating the resulting leakage (see below). For simplicity and when no ambiguity may occur, $\mathcal{F}$ is used for the flush task or step.

To represent idle processor time, define the *wait* task as

$$\mathcal{T}_\mathcal{W}(-,-,\mathcal{S}_\mathcal{W}(1,*,*))\ .$$

Similar to flush, wait is always available to be scheduled and has no period or deadline (again denoted as -). Wait also has a single step that has the minimal runtime of one time unit. However, the leakage value of wait is here denoted by $*$ since waiting does not change the memory, instead the $*$ denotes that *the leakage value of a wait step is the same as the previous step*. Similarly, the security level is also represented by $*$ because it is the same as the previous step. Again for simplicity and where no ambiguity may occur, $\mathcal{W}$ may be used in place of the wait task or step.

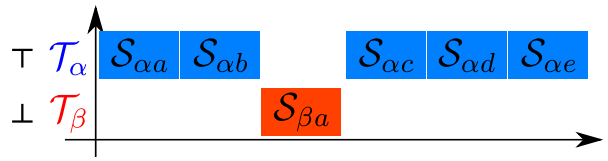
3) Traces, Solutions, and Resulting Leakage: Define a *trace* $\widetilde{\mathcal{S}} = (\mathcal{S}_1(E_1, L_1, X_1), \mathcal{S}_2(E_2, L_2, X_2), \dots)$ to be a (possibly infinite) sequence of $n \in \mathbb{N} \cup \{\infty\}$ steps that may come from any number of jobs. A trace represents a possible scheduling of jobs. Step $\mathcal{S}_1$ starts execution at time $t_1 = 0$, and each step $\mathcal{S}_i$ for $i > 1$ starts execution at time

$$t_i = \sum_{j=1}^{i-1} E_j$$

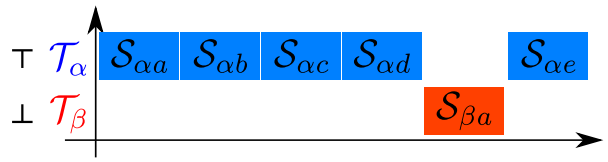
and terminates execution at time $t_i + E_i$. The notation $\widetilde{\mathcal{S}_1} +\!\!+ \widetilde{\mathcal{S}_2}$ is used to indicate concatenation of traces $\widetilde{\mathcal{S}_1}$ and $\widetilde{\mathcal{S}_2}$, and $\widetilde{\mathcal{S}} \setminus \mathcal{S}_1$ the removal of the step $\mathcal{S}_1$ from the trace $\widetilde{\mathcal{S}}$. The focus of this paper is upon *solutions*, i.e. traces of the following definition.

Definition A trace $\widetilde{\mathcal{S}}$ is a *solution* $\overline{\mathcal{S}}$ if:

1) for each job $\tau(R, A, \widehat{\mathcal{S}})$:
   a) each step in $\widehat{\mathcal{S}}$ appears in the trace $\widetilde{\mathcal{S}}$ in the order that it appears in $\widehat{\mathcal{S}}$;
   b) the first step of $\widehat{\mathcal{S}}$ does not start execution before R;



(a) Solution leaking information.



(b) Solution leaking no information.

Fig. 2: Schedulings for the processes in Fig. 1.

c) the last step of $\widetilde{\mathcal{S}}$ does not terminate execution after A;

2) each step that is not wait $\mathcal{W}$ or flush $\mathcal{F}$ appears exactly once in the trace $\widetilde{\mathcal{S}}$.

Given a set of tasks Γ, a solution $\overline{\mathcal{S}}$ is a *solution for* Γ, written $\overline{\mathcal{S}}_\Gamma$, iff $\forall \mathcal{T}_x \in \Gamma, \forall k \in \mathbb{N}_0$ then for each job $\tau_{x,k}(R_{x,k}, A_{x,k}, \widehat{\mathcal{S}_{x,k}})$ it holds that every step in $\widehat{\mathcal{S}_{x,k}}$ is in the solution $\overline{\mathcal{S}}$.

A solution $\overline{\mathcal{S}}$ is *periodic* if it periodically repeats the same sequence of steps up to job indexing. For simplicity and when no ambiguity may occur, a periodic solution may be represented by the periodically repeated sequence alone. For instance, the solutions in Fig. 3 are periodic.

Given a trace $\widetilde{\mathcal{S}}$ the *resulting leakage* $\mathscr{L}(\widetilde{\mathcal{S}})$ of trace $\widetilde{\mathcal{S}}$ represents the total amount of information that is leaked during the execution of the jobs scheduled according to the trace $\widetilde{\mathcal{S}}$.

Definition Given a trace $\widetilde{\mathcal{S}}$ composed of n steps with $n \in \mathbb{N} \cup \{\infty\}$, the *resulting leakage* $\mathscr{L}(\widetilde{\mathcal{S}})$ of the trace $\widetilde{\mathcal{S}}$ is defined inductively as follows:

- if $n \leq 1$, then $\mathscr{L}(\widetilde{\mathcal{S}}) = 0$;
- if $n > 1$ and the second step $\mathcal{S}_2$ of trace $\widetilde{\mathcal{S}}$ is $\top$, then the resulting leakage is the leakage of the trace without the first step $\mathcal{S}_1$:

$$\mathscr{L}(\widetilde{\mathcal{S}}) = \mathscr{L}(\widetilde{\mathcal{S}} \setminus \mathcal{S}_1) \quad ;$$

- if $n > 1$ and the second step $\mathcal{S}_2$ of trace $\widetilde{\mathcal{S}}$ is $\bot$, then the resulting leakage is the leakage of the trace without the first step $\mathcal{S}_1 = \mathcal{S}(E_1, L_1, X_1)$ plus the leakage value L_1 of the first step $\mathcal{S}_1$:

$$\mathscr{L}(\widetilde{\mathcal{S}}) = \mathscr{L}(\widetilde{\mathcal{S}} \setminus \mathcal{S}_1) + L_1 \quad .$$

Since every solution $\overline{\mathcal{S}}$ is a trace $\widetilde{\mathcal{S}}$, a solution's resulting leakage $\mathscr{L}(\overline{\mathcal{S}})$ is defined in the same manner.

Recall the example from Fig. 1. The solution in Fig. 2a has resulting leakage one, since `Process 2` is executed when the key is in the shared memory and so the step $\mathcal{S}_{\beta a}$ is able to access the key.

However, the solution in Fig. 2b has resulting leakage is zero, since `Process 2` is executed after the key has been wiped from the shared memory.

If a solution is periodic, then the *periodic leakage* of the solution can be calculated as follows. Given one instance of the periodically repeated sequence of steps $\overline{\mathcal{S}} = (\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_i)$, the periodic leakage is the resulting leakage of the sequence $\widetilde{\mathcal{S}}{+}{+}\mathcal{S}_1$. This is illustrated in Section III-C1 below.

C. Examples

This section presents three examples illustrating the utility of the model. Each presents a different aspect of how the model can be used to find solutions with good resulting leakage.

1) Periodic Leakage: This example illustrates leakage due to the periodic nature of tasks and how to account for this when scheduling.

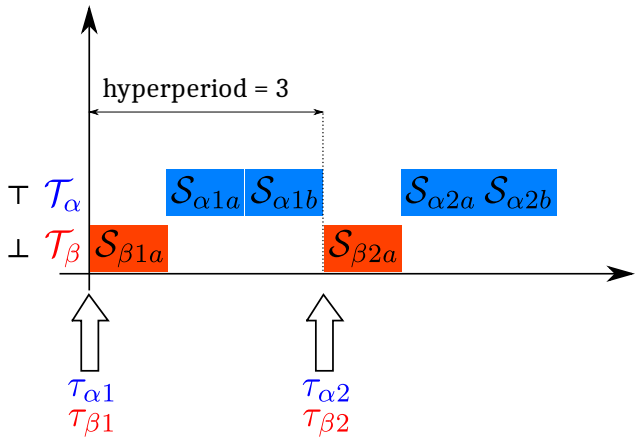
Consider two tasks: a $\top$ task $\mathcal{T}_\alpha(3, 3, \mathcal{S}_{\alpha a}(1, 0, \top), \mathcal{S}_{\alpha b}(1, 4, \top))$ and a $\bot$ task $\mathcal{T}_\beta(3, 3, \mathcal{S}_{\beta a}(1, 0, \bot))$. The goal is to find a solution with minimal (here zero) resulting leakage.

Two periodic solutions to these tasks are depicted in Fig. 3. Note that the $\top$ step $\mathcal{S}_{\alpha 1a}$ has leakage value zero, so even if $\mathcal{S}_{\alpha 1a}$ is followed by the $\bot$ step $\mathcal{S}_{\beta 1a}$ this does not increase the resulting leakage. Thus both periodic solutions have a resulting leakage of zero within their periodically repeated sequence (here corresponding to their hyperperiod). However, when the periodically repeated sequence is repeated, the periodic solution in Fig. 3a has non-zero periodic leakage, since at time four the $\top$ step with leakage value four $\mathcal{S}_{\alpha 1b}$ is followed by the $\bot$ step $\mathcal{S}_{\beta 2a}$ on periodic scheduling. Hence, only the periodic solution in Fig. 3b has periodic leakage zero.

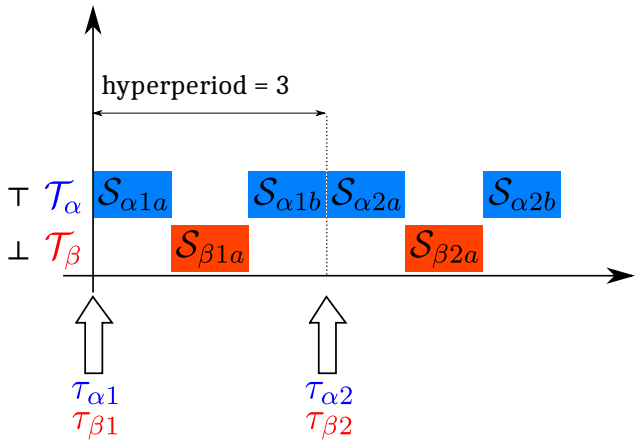
2) Waiting can Reduce Leakage: This example illustrates the utility of making $\mathcal{W}$ available to the scheduler. In some cases exploiting $\mathcal{W}$ reduces the resulting leakage of a solution.

Consider two tasks: a $\top$ task $\mathcal{T}_\alpha(6, 5, \mathcal{S}_{\alpha a}(1, 0, \top), \mathcal{S}_{\alpha b}(1, 4, \top))$ and a $\bot$ task $\mathcal{T}_\beta(3, 2, \mathcal{S}_{\beta a}(1, 0, \bot))$. Again the goal is to find a solution with the minimal (zero) resulting leakage.

One periodic solution to this example with zero resulting leakage is presented in Fig. 4. The periodic solution exploits $\mathcal{W}$ to have the two $\bot$ steps executed together (with only $\mathcal{W}$ in between). This allows the $\top$ step $\mathcal{S}_{\alpha 1b}$ (with positive leakage value) to be scheduled after the last $\bot$ step in the periodically repeated sequence (again corresponding here to the hyperperiod) and still meet its deadline. Observe that since $\mathcal{S}_{\alpha 2a}$ has zero leakage value, the periodic solution has zero periodic leakage. This periodic solution with zero periodic leakage would not be possible if $\mathcal{W}$ was not available to be scheduled at any time (in particular as an alternative to scheduling the step $\mathcal{S}_{\alpha 1b}$), since without $\mathcal{W}$ then $\mathcal{S}_{\alpha 1b}$ would be scheduled before $\mathcal{S}_{\beta 2a}$ and thus increase periodic leakage.



(a) Periodic solution leaking information.



(b) Periodic solution leaking no information.

Fig. 3: Leakage between hyperperiods.

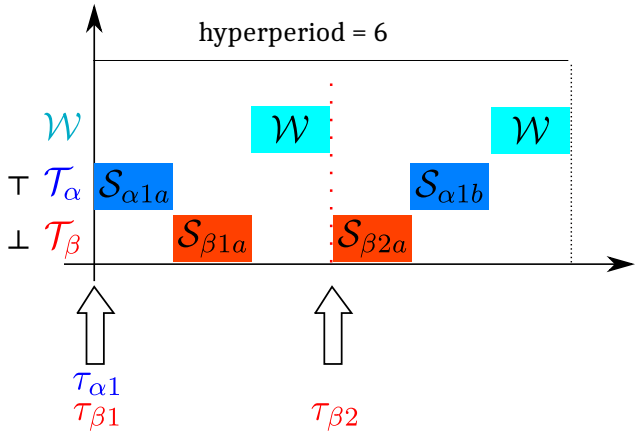


Fig. 4: Waiting can reduce leakage.

3) Periodic Flush: Since finding a solution for a set of tasks is generally best solved in a periodic manner, it is possible to exploit this periodic nature when constructing the solution. For example, the total amount of time units not used by jobs can be calculated, and then these time units can be used to consider adding $\mathcal{F}$. Typically, such free time units would be fragmented inside the solution. However, with this information, the scheduler can use sufficiently long empty spaces (or create them) to schedule $\mathcal{F}$. Hence, even if it may not be possible to flush the memory after each $\top$ step followed by a $\bot$ step, some additional $\mathcal{F}$ can be scheduled to reduce the solution's resulting leakage while maintaining schedulability.

For example, consider two tasks: a $\top$ task $\mathcal{T}_\alpha(8, 8, \mathcal{S}_{\alpha a}(1, 5, \top), \mathcal{S}_{\alpha b}(1, 1, \top), \mathcal{S}_{\alpha c}(1, 6, \top), \mathcal{S}_{\alpha d}(1, 4, \top))$ and a $\bot$ task $\mathcal{T}_\beta(3, 3, \mathcal{S}_{\beta a}(1, 0, \bot))$. Here let the execution time of $\mathcal{F}$ be two, i.e. $E_\mathcal{F} = 2$. Consider the scheduling of the two tasks over their hyperperiod of twenty-four time units (when developing periodic solutions, the hyperperiod is a convenient choice since periodicity is guaranteed for the hyperperiod). A periodic solution can be seen in Fig. 5.

No solution with resulting leakage zero exists. Further, it is not possible to schedule the jobs by inserting a $\mathcal{F}$ after every $\top$ step followed by a $\bot$ step since this would not be schedulable (this is the state of the art as in [13]). However, the periodic solution in Fig. 5 achieves a low periodic leakage of three per hyperperiod while maintaining schedulability by adding two $\mathcal{F}$ steps to minimize the periodic leakage.

IV. Our Approach

The overarching goal of the approach proposed in this paper is to produce a solution with low resulting leakage for a given set of tasks.

To achieve this, standard offline scheduling algorithms are extended with a *preprocessing* and a *postprocessing* phase. The preprocessing phase transforms a set of tasks Γ into a set of preprocessed tasks Γ'. Then scheduling is applied to Γ' obtaining a solution $\overline{\mathcal{S}'}_{\Gamma'}$ for Γ'. Finally, the postprocessing phase transforms the solution $\overline{\mathcal{S}'}_{\Gamma'}$ into a postprocessed solution $\overline{\mathcal{S}''}_{\Gamma'}$.

Both the pre- and postprocessing phases can affect the desired solution $\overline{\mathcal{S}''}_{\Gamma'}$, here with the goal of reducing the resulting leakage.

The rest of this section presents various heuristic algorithms used for the results (see Section V). The scheduling algorithms considered are EDF and LSF. The rest of this section focuses in particular upon the pre- and postprocessors. The division in phases creates a modular and compositional approach, allowing for a better comparison of different pre- and postprocessors.

A. Preprocessing

Preprocessors are algorithms that take a set of tasks Γ and produce a set of tasks Γ' to be scheduled. This paper considers preprocessors that attempt to "merge" adjacent steps with the same security level within each task in Γ. The merged step has the sum of the execution times of the merged steps, the leakage value of the last merged step, and the same security level as the merged steps. For instance, the steps $\mathcal{S}_{\alpha a}(1, 0, \top)$ and $\mathcal{S}_{\alpha b}(1, 4, \top)$ could be merged producing the step $\mathcal{S}_{\alpha a'}(2, 4, \top)$. The rest of this section presents three preprocessing algorithms that exploit merging.

Total Merge: The Total Merge algorithm merges all the steps in a task into a single step, as detailed in Algorithm 1. The merging is achieved by starting with a step that has execution time and leakage value zero. The execution time for each other step in the task is then added, and the leakage value

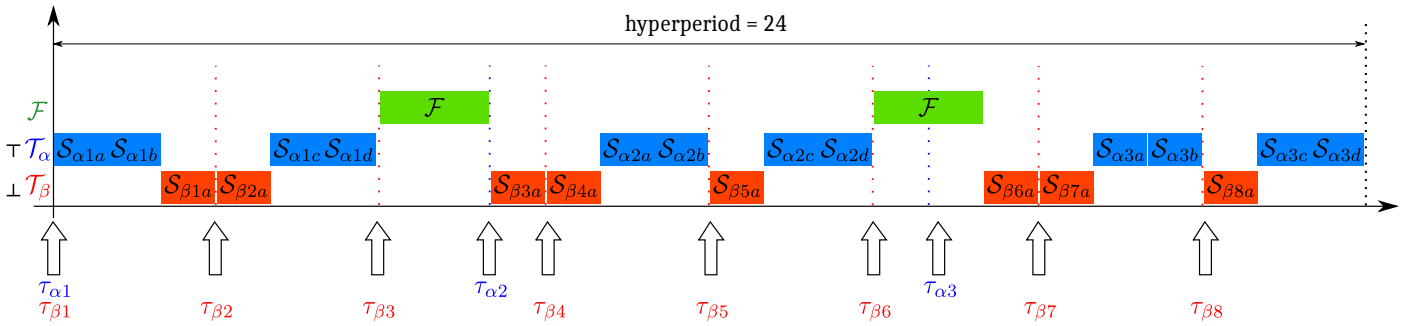


Fig. 5: Hyperperiodic flush.

Data: task $\mathcal{T}(P, D, \widehat{\mathcal{S}})$
Result: processed task $\mathcal{T}'$
$E_T = 0$
$L_T = 0$
$X_T = \bot$
for $i = 1$ *to* $|\widehat{\mathcal{S}}|$ **do**
  let $\mathcal{S}(E_i, L_i, X_i) = \mathcal{S}_i$
  $E_T = E_T + E_i$
  $L_T = L_i$
  $X_T = X_i$
end
Return $\mathcal{T}' = \mathcal{T}(P, D, \mathcal{S}_T(E_T, L_T, X_T))$

Algorithm 1: Total Merge Preprocessor

from the last step being merged is preserved. The security level is set to that of the last step (this is reasonable here since all steps within a task share the same security level, for modeling mixed security levels see Section VI-A2). Finally, the processed task uses this single merged step as the only step in its sequence of steps.

One-Step Merge: The One-Step Merge algorithm attempts to merge pairs of adjacent steps. Adjacent pairs are merged iff the leakage of the former step is higher than the leakage of the latter. This is achieved by iterating through the steps $\mathcal{S}_i$ of the task. If $L_i > L_{i+1}$, then the steps $\mathcal{S}_i$ and $\mathcal{S}_{i+1}$ are merged. Otherwise, $\mathcal{S}_i$ is maintained unchanged. This algorithm generates a new sequence of steps $\widehat{\mathcal{S}'}$, that are then used in the processed task. Details can be seen in Algorithm 2.

n-Step Merge: A straightforward extension to the One-Step Merge algorithm is to allow merging of any number of steps. This appears in the results as the *n*-Step Merge algorithm, and is a straightforward modification of Algorithm 2.

B. Postprocessing

Postprocessing algorithms take one solution and produce another solution. This can be done by any possible manipulation of the steps within the original solution $\overline{\mathcal{S}'}_\Gamma$ to produce the new solution $\overline{\mathcal{S}''}_\Gamma$ that does not break the property of being a solution for Γ. The rest of this section presents four such postprocessors.

Add Flush: The Add Flush algorithm replaces sequences of $\mathcal{W}$ with $\mathcal{F}$ where possible. This is defined in Algorithm 3. Add Flush operates by finding sequences of $\mathcal{W}$

Data: task $\mathcal{T}(P, D, \widehat{\mathcal{S}})$
Result: processed task $\mathcal{T}'$
$\widehat{\mathcal{S}'} = \emptyset$
$i = 1$
while $i < |\widehat{\mathcal{S}}| + 1$ **do**
  let $\mathcal{S}(E_i, L_i, X_i) = \mathcal{S}_i$
  let $\mathcal{S}(E_{i+1}, L_{i+1}, X_{i+1}) = \mathcal{S}_{i+1}$
  if $L_{i+1} < L_i$ **then**
    $\widehat{\mathcal{S}'} = \widehat{\mathcal{S}'} +\!\!+ \mathcal{S}(E_i + E_{i+1}, L_{i+1}, X_{i+1})$
    $i = i + 2$
  else
    $\widehat{\mathcal{S}'} = \widehat{\mathcal{S}'} +\!\!+ \mathcal{S}_i$
    $i = i + 1$
  end
end
Return $\mathcal{T}' = \mathcal{T}(P, D, \widehat{\mathcal{S}'})$

Algorithm 2: One-Step Merge Preprocessor

whose length is greater than or equal the execution time of $\mathcal{F}$. If such a sequence is found, a $\mathcal{F}$ is added to the produced solution instead of the initial sequence of $\mathcal{W}$ with execution time equal to the $\mathcal{F}$. Any remaining $\mathcal{W}$ in the solution are maintained as they were before.

Swap: The Swap algorithm attempts to reduce the resulting leakage by swapping steps within the solution as shown in Algorithm 4. Swap works by considering each step $\mathcal{S}_i$. Then each possible swap $[\mathcal{S}_i \leftrightarrow \mathcal{S}_j]$ between the step $\mathcal{S}_i$ and a following step $\mathcal{S}_j$ is considered. If the trace with this swap applied has less resulting leakage and is still a solution, then this solution $[\mathcal{S}_i \leftrightarrow \mathcal{S}_j]\overline{\mathcal{S}}$ is kept as the best possible solution so far. Finally, once all possible swaps have been considered, the best swap to the solution is applied and i is incremented.

Move: The Move algorithm moves one step to a new position in the solution. Move works in the same manner as the Swap postprocessor, except instead of swapping $[\mathcal{S}_i \leftrightarrow \mathcal{S}_j]\overline{\mathcal{S}}$ the steps $\mathcal{S}_i$ and $\mathcal{S}_j$, the move $[\mathcal{S}_i \longrightarrow \mathcal{S}_j]\overline{\mathcal{S}}$ moves the step $\mathcal{S}_i$ to be after $\mathcal{S}_j$. For example:

$$[\mathcal{S}_1 \longrightarrow \mathcal{S}_3]\mathcal{S}_a, \mathcal{S}_b, \mathcal{S}_c \quad = \quad \mathcal{S}_b, \mathcal{S}_c, \mathcal{S}_a$$

where the first step $\mathcal{S}_a$ is moved to be after the third step $\mathcal{S}_c$. The rest of the algorithm is the same as Swap, finding the best possible move and ensuring the trace after the move is a

Data: solution $\overline{\mathcal{S}}$, and wait $\mathcal{W}$
Result: solution $\overline{\mathcal{S}'}$
1 $\overline{\mathcal{S}'} = \emptyset$
2 $i = 1$
3 **while** $i < |\overline{\mathcal{S}}| + 1$ **do**
4 **if** $\mathcal{S}_i == \mathcal{W}$ **then**
5 $j = 1$
6 **while** $i + j < |\overline{\mathcal{S}}|$ && $\mathcal{S}_{i+j} == \mathcal{W}$ **do**
7 $j = j + 1$
8 **end**
9 $i = i + j$
10 **if** $j \geq E_{\mathcal{F}}$ **then**
11 $\overline{\mathcal{S}'} = \overline{\mathcal{S}'} ++ \mathcal{F}$
12 $j = j - E_{\mathcal{F}}$
13 **end**
14 **while** $j > 0$ **do**
15 $\overline{\mathcal{S}'} = \overline{\mathcal{S}'} ++ \mathcal{W}$
16 $j = j - 1$
17 **end**
18 **else**
19 $\overline{\mathcal{S}'} = \overline{\mathcal{S}'} ++ \mathcal{S}_i$
20 $i = i + 1$
21 **end**
22 **end**
23 Return $\overline{\mathcal{S}'}$

Algorithm 3: Add Flush Postprocessor

Data: solution $\overline{\mathcal{S}}$
Result: solution $\overline{\mathcal{S}'}$
1 **for** $i = 1$ *to* $|\overline{\mathcal{S}}|$ **do**
2 $\overline{\mathcal{S}'} = \overline{\mathcal{S}}$
3 **for** $j = i$ *to* $|\overline{\mathcal{S}}|$ **do**
4 $\overline{\mathcal{S}''} = [\mathcal{S}_i \leftrightarrow \mathcal{S}_j]\overline{\mathcal{S}}$
5 **if** $\mathscr{L}(\overline{\mathcal{S}''}) < \mathscr{L}(\overline{\mathcal{S}'})$ && *isSolution*$(\overline{\mathcal{S}''})$ **then**
6 $\overline{\mathcal{S}'} = \overline{\mathcal{S}''}$
7 **end**
8 **end**
9 $\overline{\mathcal{S}} = \overline{\mathcal{S}'}$
10 **end**
11 Return $\overline{\mathcal{S}}$

Algorithm 4: Swap Postprocessor

solution. The algorithm is identical to Algorithm 4 substituting $[\mathcal{S}_i \leftrightarrow \mathcal{S}_j]\overline{\mathcal{S}}$ with $[\mathcal{S}_i \longrightarrow \mathcal{S}_j]\overline{\mathcal{S}}$ in Line 4.

1-Swap: Observe that if only swapping or moving with the following step is considered, that is $[\mathcal{S}_i \leftrightarrow \mathcal{S}_{i+1}]$ or $[\mathcal{S}_i \longrightarrow \mathcal{S}_{i+1}]$, then the swap and move postprocessors coincide. This postprocessor is denoted as 1-Swap in the results.

C. Modeling Existing Work

Section II-C recalls prior work proposing approaches regarding confidentiality when tasks with different security levels are sharing resources. This section demonstrates how some of this work can be represented in our model.

Mohan et al. [13] also propose to add a flush task in order to remove confidential information from the shared memory. In their model each task has only a single step that performs all operations of that task. This is equivalent to applying the Total Merge preprocessor to each task in the model proposed here. Mohan et al. schedule a memory flush after any $\top$ step that is to be followed by a $\bot$ step. This could be modeled by a postprocessor implementing the same approach.

Their approach guarantees that any solution has zero resulting leakage. However, the weakness of their approach is that schedulability is often not preserved, e.g., the tasks in Fig. 3 would not be schedulable. Since Total Merge already yields unacceptable schedulability results (as shown in Section V) and such a postprocessor would degrade schedulability even further, this was not considered in the experimental results.

V. Experimental Results

This section discusses the results obtained by running experiments with the preprocessing, scheduling, and postprocessing algorithms in this paper. The experiments were conducted by using approximately 30,000 randomly generated sets of tasks[2], and then testing each possible combination of one preprocessing, one scheduling, and one postprocessing algorithm. Each set of tasks consists of 2 to 6 tasks with at least one $\top$ task and one $\bot$ task, with each task having 1 to 8 steps, and each step execution time from 1 to 5. $\top$ steps have a 0.7 probability of having a leakage value from 1 to 8, and a leakage value of 0 otherwise. Sets of tasks with a hyperperiod over 5000 have been discarded to reduce testing time. The code to perform the tests and implement the preprocessing, scheduling, and postprocessing is written in Java 1.8[3], and all experiments conducted on a Linux 3.13 64-bit kernel on an Intel Core i7-3720QM 2.60GHz CPU with 8GB of RAM. The code used to run these experiments is available upon request. The rest of this section discusses the outcomes of the experiments.

The first point of interest is the schedulability of the set of tasks used in each experiment. Merging task steps in a preprocessor can make a set of tasks unschedulable, and the EDF and LSF scheduling algorithms are not equally able to find solutions. The failure percentage for each combination of preprocessing and scheduling algorithm is shown in Fig. 6.

Fig. 6 clearly shows that greater merging of steps leads to more schedulability failures. In particular, indicating that Total Merge is not an effective algorithm to use in practice despite being considered the current state of the art [13], [14]. This is a strong motivation for the approach presented in this work to consider fine-grained preprocessing and preemption of tasks. Due to its high failure rate, Total Merge will not be considered further in this paper.

Fig. 6 also shows that, for all preprocessing algorithms, EDF performs better for schedulability than LSF. (This is expected since EDF is guaranteed to find a solution if the tasks are schedulable, while LSF is not.) The two scheduling algorithms produce almost the same results for every other measure tested, so the rest of this paper shall present only experimental results using the EDF scheduling algorithm.

[2] 30,000 sets of tasks were generated, 22 were discarded as unschedulable.
[3] Code is available via git at:
https://scm.gforge.inria.fr/anonscm/git/secleakpublic/secleakpublic.git

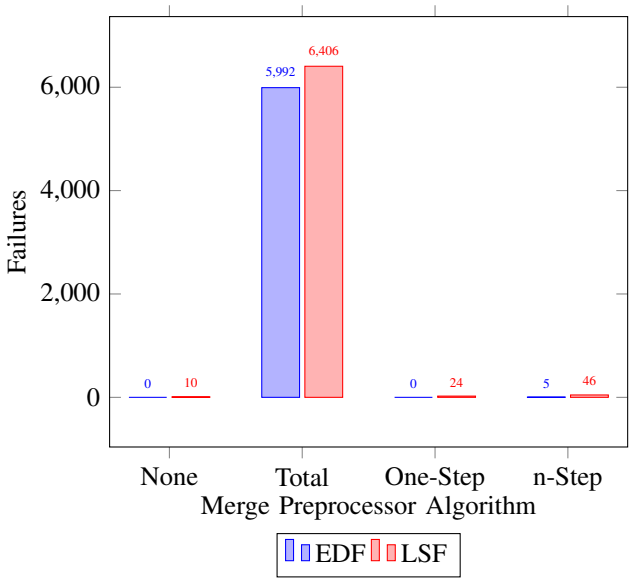


Fig. 6: Number of failures for each combination of preprocessor and scheduling algorithm, out of ~30,000 experiments. Note that Total Merge, corresponding to the state of the art [13], [14], fails ~20% of the time.

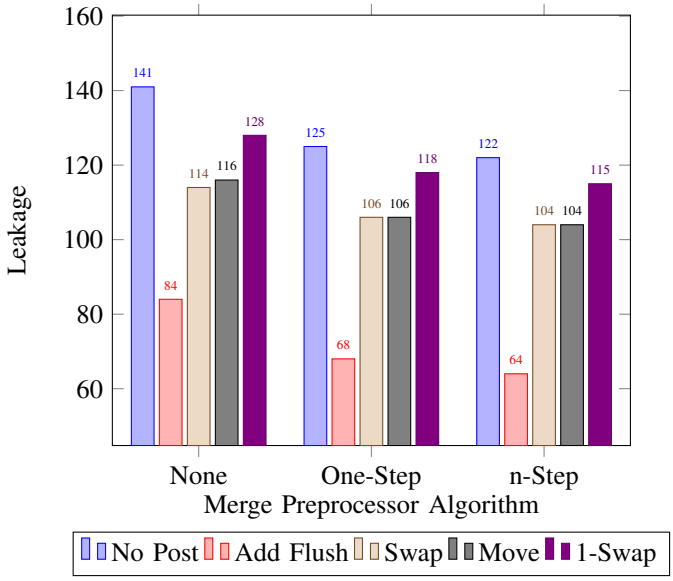


Fig. 7: Information leakage of the solutions for each combination of pre- and postprocessor (except Total Merge) using the EDF scheduling algorithm.

Comparing the experimental results from postprocessing algorithms, the average resulting leakages for each combination of pre- and postprocessor is shown in Fig. 7, while the average running times are shown in Table I.

Preprocessor Merge	**Postprocessor**				
	None	**Add Flush**	**Swap**	**Move**	**1-Swap**
None	2	116	1919	1903	190
One-Step	1	93	1567	1489	149
n-Step	1	88	1486	1404	141

TABLE I: Average execution time (in ms) for each combination of pre- and postprocessor (except Total Merge) using the EDF scheduling algorithm.

As expected, solutions without any postprocessing produce the highest resulting leakage. The best resulting leakage is obtained by the Add Flush algorithm. Note that merging preprocessors reduce total time, since they reduce the number of steps that the scheduler has to schedule.

1-Swap slightly reduces the resulting leakage, however Table I shows that it is significantly more expensive than the scheduling operation, so 1-Swap could be applied after Add Flush only if the cost is acceptable. Swap and Move do not reduce the resulting leakage significantly more than 1-Swap and are significantly more expensive to compute, indicating that simple heuristics may be sufficient in practice.

VI. Discussion

This section discusses some of the choices made in this paper and their implications.

A. On Leakage

There are several choices made with respect to leakage where other choices are possible.

1) Different Leakage Models: The model of resulting leakage here is based upon the last leakage value of a step prior to a following $\perp$ step. This is simple and easy to compute, but more complex approaches could yield better matches to different scenarios. A different approach could be cumulative with respect to resulting leakage, since there is no strong reason to believe that any given step would wipe all other confidential information from the shared memory. Thus, resulting leakage could be cumulative unless confidential information is known to be wiped in specific steps (although this would tend to over approximate significantly in practice). An alternative would be to use an algebraic approach, similar to that defined for $\mathcal{W}$, that depends on the leakage value of the previous step. This could then include many functions on the leakage values of prior steps, modeling partial over-writing, deletion, or non-interference between confidential information.

2) $\perp$ Steps with Positive Leakage Value: For simplicity in this paper, $\perp$ steps have always been presented with zero leakage value. However, there is no reason to believe that a $\perp$ step would never have access to confidential information (e.g. when this confidential information could have come from an earlier leak). Thus, $\perp$ steps could also have positive (or more complex as in Section VI-A1) leakage values.

In addition to modeling confidential information passing through $\perp$ steps to yield leakage, $\perp$ steps with positive leakage values could also come from merging steps. For example, merging two steps (a $\perp$ step (with zero leakage value) and a following $\top$ step with positive leakage value) could yield a $\perp$ step with positive leakage value. This would naturally capture the intended resulting leakage in a trace, without having to add extra kinds of security levels (such as $\top$-to-$\perp$ and $\perp$-to-$\top$).

B. On the Division of Scheduling into Three Phases

The division into three phases is to separate out distinct parts of an overall scheduling from tasks to a solution. This approach allows for separation conceptually of different phases, and also for composition of simple algorithms in the pre- and postprocessing phases. For example, a postprocessor could

move steps in a solution around to maximize contiguous $\mathcal{W}$s and then be composed with the Add Flush postprocessor to improve the resulting leakage further. This also allows different strategies to be employed in different phases, including strategies with different goals. For example, processors for resulting leakage minimization and energy consumption could be combined during pre- or postprocessing (or both).

VII. Conclusions and Future Work

In a system with shared resources, the security of confidential information is a major concern. This paper presents a model that allows reasoning about the leakage of such confidential information via shared resources. The presented model extends the well-known workflow model to support fine-grained preemption and confidentiality. This allows confidentiality to be accounted for by scheduling approaches. Confidentiality is addressed by quantifying the amount of information leaked by the system. The model does not restrict the semantic meaning of the leakage values, and can be used with different leakage models.

Scheduling in this new model is then considered using pre- and postprocessors. The preprocessors are compositional and implement operations upon tasks that are to be scheduled. Similarly, postprocessors are compositional and operate on scheduling solutions. These can be combined to allow for scheduling that exploits different techniques and approaches, including focusing on different aspects of the overall problem. For example, some on leakage, others on schedulability, and yet others on different concerns such as power usage.

Several pre- and postprocessing heuristic algorithms are presented that can operate on the model. These are focused upon improving resulting leakage, but the principles can be adapted to other areas as well.

The experimental results show that the Total Merge algorithm that models the coarse-grained approach in the state of the art performs poorly, and indeed often fails to find a scheduling solution. Of the other preprocessors, it is clear that there is a trade-off between: reducing resulting leakage, and achieving schedulability. Small amounts of merging steps (One-Step Merge) prove beneficial without schedulability cost. For the postprocessors, Add Flush performs the best with minimum cost, suggesting that even simple heuristics can provide good results with minimal extra cost to scheduling. This also implies that non-realtime systems could benefit from even simple heuristics to improve confidentiality.

Considering future work, this paper introduces a model to consider confidentiality as a resource. Hence, it would be interesting to generalize this to a multi-resource approach, where schedulers consider confidentiality, energy consumption, schedulability, etc. at the same time. Another direction would be to consider theoretical complexity. A possible extension of this work could be, given a set of tasks, determining the complexity of finding a solution: (1) with zero resulting leakage, (2) with at most a given resulting leakage, and (3) with the minimal resulting leakage.

References

[1] M. S. Alvim, K. Chatzikokolakis, C. Palamidessi, and G. Smith. Measuring information leakage using generalized gain functions. In S. Chong, editor, *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 265–279. IEEE Computer Society, 2012.

[2] M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 141–153. IEEE Computer Society, 2009.

[3] A. Benoit, U. V. Çatalyürek, Y. Robert, and E. Saule. A survey of pipelined workflow scheduling: Models and algorithms. *ACM Comput. Surv.*, 45(4):50:1–50:36, Aug. 2013.

[4] F. Biondi, A. Legay, P. Malacaria, and A. Wasowski. Quantifying information leakage of randomized protocols. *Theor. Comput. Sci.*, 597:62–87, 2015.

[5] F. Biondi, A. Legay, L. Traonouez, and A. Wasowski. QUAIL: A quantitative security analyzer for imperative code. In N. Sharygina and H. Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 702–707. Springer, 2013.

[6] R. Chadha, U. Mathur, and S. Schwoon. Computing information flow using symbolic model-checking. In V. Raman and S. P. Suresh, editors, *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India*, volume 29 of *LIPIcs*, pages 505–516. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.

[7] T. Chothia, Y. Kawamoto, and C. Novakovic. Leakwatch: Estimating information leakage from java programs. In M. Kutylowski and J. Vaidya, editors, *Computer Security - ESORICS 2014 - 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II*, volume 8713 of *Lecture Notes in Computer Science*, pages 219–236. Springer, 2014.

[8] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet dossier, 2011.

[9] R. L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 45(9):1563–1581, Nov 1966.

[10] J. Heusser and P. Malacaria. Quantifying information leaks in software. In C. Gates, M. Franz, and J. P. McDermott, editors, *Twenty-Sixth Annual Computer Security Applications Conference, ACSAC 2010, Austin, Texas, USA, 6-10 December 2010*, pages 261–269. ACM, 2010.

[11] J. H. Kim, A. Legay, K. G. Larsen, M. Mikucionis, and B. Nielsen. Resource-parameterized timing analysis of real-time systems. In N. Piterman, editor, *Hardware and Software: Verification and Testing - 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings*, volume 9434 of *Lecture Notes in Computer Science*, pages 190–205. Springer, 2015.

[12] J. H. Kim, A. Legay, L. Traonouez, A. Boudjadar, U. Nyman, K. G. Larsen, I. Lee, and J. Choi. Optimizing the resource requirements of hierarchical scheduling systems. *SIGBED Review*, 13(3):41–48, 2016.

[13] S. Mohan, M. Yoon, R. Pellizzoni, and R. Bobba. Real-time systems security through scheduler constraints. In *26th Euromicro Conference on Real-Time Systems, ECRTS 2014, Madrid, Spain, July 8-11, 2014*, pages 129–140. IEEE Computer Society, 2014.

[14] R. Pellizzoni, N. Paryab, M. Yoon, S. Bak, S. Mohan, and R. Bobba. A generalized model for preventing information leakage in hard real-time systems. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium, Seattle, WA, USA, April 13-16, 2015*, pages 271–282. IEEE Computer Society, 2015.

[15] J. Son and J. Alves-Foss. Covert timing channel capacity of rate monotonic real-time scheduling algorithm in MLS systems. In S. Rajasekaran, editor, *Proceedings of the Third IASTED International Conference on Communication, Network, and Information Security, October 9-11, 2006, Cambridge, MA, USA*, pages 13–18. IASTED/ACTA Press, 2006.

[16] S. H. Son, R. Mukkamala, and R. David. Integrating security and real-time requirements using covert channel capacity. *IEEE Trans. Knowl. Data Eng.*, 12(6):865–879, 2000.

[17] C. G. Val, M. A. Enescu, S. Bayless, W. Aiello, and A. J. Hu. Precisely measuring quantitative information flow: 10k lines of code and beyond. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 31–46. IEEE, 2016.