Report on monitoring data minimisation

Abstract. Data minimisation is a privacy enhancing principle, stating that personal data collected should be no more than necessary for the specific purpose consented by the user. Checking that a program satisfies the data minimisation principle is not easy, even for the simple case when considering deterministic programs-as-functions. In this paper we prove (im)possibility results concerning runtime monitoring of (non-)minimality for deterministic programs both when the program has one input source (monolithic) and for the more general case when inputs come from independent sources (distributed case). We propose monitoring mechanisms where a monitor observes the inputs and the outputs of a program, to detect violation of data minimisation policies. We show that monitorability of non-minimality is decidable only for specific cases while it is undecidable in general for different notions of minimality. That said, we show that under certain conditions monitorability is decidable and we provide an algorithm and a bound to check such properties in a pre-deployment controlled environment, also being able to compute a minimiser for the given program. Finally, we provide a proof-of-concept implementation for both offline and online monitoring and apply that to some case studies.

1 Introduction

According to the Article 5 of the *General Data Protection Regulation* proposal (GDPR —EU–2016/679, adopted on 27 April 2016 and entering into application on 25 May 2018), "Personal data must be adequate, relevant, and limited to the minimum necessary in relation to the purposes for which they are processed" [8]. While determining what is "adequate" and "relevant" might seem difficult given the inherent imprecision of the terms, identifying what is "minimum necessary in relation to the purpose" seems to be easier. We could understand this principle in different ways, and we discuss below a couple of possible interpretations. We do so by having in mind that our objective is to find a way to enforce, or at least detect, when the *data minimisation* principle is (not) satisfied from a technical point of view, and in particular by using a language-based approach.

One way to understand minimisation is on how the data is *used*, that is we could consider ways to identify how the input data is used in the program, for which *purposes*. This would imply that we look inside the program and track the usage of the data by performing static analysis techniques like tainting, def-use, information flow, etc. For that we need of course to have a precise definition of what purpose means and a way to check that the intended purpose matches with the real purpose under which data will be processed.

Another way to see minimisation is by considering when and how the data is *collected* and only allow the collection of data that is actually needed to compute what is required to achieve the given purpose. In this case we could consider that the "purpose" is given by the specification of the program.

In this paper we take the second view, following [1]. This kind of data minimisation calls for semantic foundations to determine whether or not a program could run equally well with less personal data input. Indeed, syntax-driven techniques do not give any information about the semantic "necessity" as meant by the proposal. It is to be noted that this principle exists in other regulations and is sometimes referred to as "collection limitation" when it focuses on the particular step of collecting data. This is for instance the case of the *Fair Information Practice Principles* (FIPPs) [16] in USA, and the *Guidelines on the Protection of Privacy and Transborder Flows of Personal Data* [14] proposed by the Organisation for Economic Co-operation and Development (OECD).

Determining the quantity of information actually needed for a given purpose requires an analysis of the program. Following the approach detailed in [15], it is possible to quantify the amount of information input to a program, the amount of information semantically used to compute the output, and the amount of input information not semantically used. If we consider data minimisation from the regulatory point of view, the input data not semantically used in the program should not be collected (and thus not processed). This is because, unlike the case in [15] where the attacker only has access to the outputs,

```
1 input(salary)
2 benefits := (salary < 10000)
3 output(benefits)
```

Fig. 1: Running example P_{bl} to compute a benefits level.

the attacker is here the data processor[1] itself, having then the possibility to also exploit the inputs. As a consequence, the attacker knows all the information available after the input is collected (before the program execution).

Given that *input data = necessary data + extra data*, and since the program should execute equally well without any extra data, we have that *input data* $\geq$ *necessary data*. The goal of the *data minimisation process* is thus to minimise the input data so only what is necessary is given to the program. Whenever the input data exactly matches what is necessary we may say that the minimisation is *perfect*. Perfect minimisation is, however, difficult to achieve in general among other things because it is not trivial to exactly determine what is the input needed to compute each possible output [1]. That said, it could be possible to achieve *some degree* of data minimisation, which though not optimal could still be considered useful (we could at least state that the program under consideration does use more input data than needed), or to be able to detect whether the data minimisation principle is violated during execution of the program.

When dealing with data minimisation we could ask ourselves the following two questions. First, "does this program perfectly respect the data minimisation principle?". If the answer is Yes, then we are happy and we could certify that the program is in conformance with the regulation. If the answer is No then we should ask ourselves whether the program could be somehow transformed so it satisfies the minimisation principle. Or, instead we could ask ourselves "is it possible to get a data minimiser such that it generates only the necessary inputs for the given program?" By trying to answer the latter, instead of trying to transform the original program, a procedure could be given to achieve data minimisation. This is exactly the solution proposed in [1] based on the generation of another program (called the *data minimiser*) that filters the input given to the original program so that it is run on a smaller set of input data (and without changing its behaviour).

Let us consider a simple program to exemplify this notion of data minimisation and sketch our solution (see Figure 1). The purpose of the program is to compute the *benefit level* of employees depending on their salary (assumed to be between $ 0 and $ 100000). For the sake of simplicity, in what follows we do not assume any particular distribution over their domain for the inputs, driving the analysis on worst-case assumptions. A quick analysis of this small program clearly shows that the range of the output is $\{\mathsf{false}, \mathsf{true}\}$, and consequently the data processor does not need to precisely know the real salaries of the employees to determine the benefit level. In principle each employee should be able to give any number between 0 and 9999 as input if they are eligible to the benefits, and any number between 10000 and 100000 otherwise, without disclosing their real salaries.

Antignac et. al. [1] defined the concept of data minimiser as a pre-processor that filters the input of the given program in such a way that the functionality of the program does not change but it only receives data that is necessary and sufficient for the intended computation. From there they derived the concept of data minimisation and they showed how to obtain data minimisers for both the *monolithic* case (only one source of input) and the *distributed* case (more than one, independent, source of inputs). The latter is clearly a semi-decision procedure given the underlying undecidability result of the problem (it reduces to computing the kernel of a function which semantically denotes the program). The approach to obtain minimisers is based on a combination of using a symbolic execution engine and a SAT solver. The proof-of-concept implementation provided in that paper only works (automatically) for simple programs (without loop, recursion, nor call to libraries). By providing loop invariants, additional specification for libraries, etc., it is possible to get a semi-automatic way to get minimisers though the manual effort hampers a large scale use of this approach.

Being one of the first papers on data minimisation, we welcome the effort in giving a formal definition of minimisation and the study of some of its properties. The approach is however quite limited, mostly in

[1] "Data controllers" and "data processors" are legal roles used to define obligations and liabilities of the parties. We indistinctly use the term "data processor" in this paper as we are interested in designating the party that technically processes the data.

what concerns the generation of data minimisers and even checking whether a given program is minimal or not.

In this paper we take those results further by consider a more practical approach. Knowing that it is in general impossible to compute data minimisers for arbitrary programs by static analysis, we consider here a *runtime* approach. Our starting point is the definition of data minimisation as a modification of Cohen's notion of *strong dependency* [6]. As in [1] we consider both the monolithic and the distributed case. We define the notion of runtime monitors for both cases extracting them from the definition of monolithic and (weak) distributed (non-)minimality. We consider two different scenarios: (i) After the program has been deployed, we perform *online monitoring* without any knowledge about how the environment will produce the inputs; (ii) Before deployment, where we could perform both *offline monitoring* (where traces are produced beforehand and fed into the monitor), or *controlled online monitoring* where we produce the inputs in a systematic way in order to capture as much of the input domain as possible (all in some cases) in a way reminiscent to the approaches developed for test cases generation.

As for most non-interference properties, checking whether a given program satisfies the data minimality principle implies checking a *hyper-property* [5, 10], that is a property over set of traces and not over a single trace. For monitoring hyperproperties we need to consider multiple executions of the program and this has been shown to be computationally hard [4]. In general it is not possible to reduce hyper-property checking into checking over a single trace, but for (non-)minimality this is the case. As a consequence, we transform the problem of checking a hyper-property over a given program into checking a property over repeated executions of the program (program-in-a-loop) so the monitoring problem can be reduced to the analysis of a single trace.

We briefly sketch here our approach over the example shown in Figure 1, focusing only on how to check non-minimality using online monitoring, and how we use a slight variation of the same procedure before deployment to obtain a minimiser[2]. Our monitor is a simple program taking input/output pairs and checking whether for different inputs the program gives the same output. Informally, if the monitor finds two pairs (i_1, o_1) and (i_2, o_2) such that $i_1 \neq i_2$ but $o_1 = o_2$ then this would be a violation of (monolithic) minimality (and such pairs would provide a witness for non-minimality)[3]. For this example, if the monitor observes the following two executions: $(5000, \mathsf{true})$, $(11000, \mathsf{false})$, it cannot conclude anything given that no same output for different inputs has been produced. However, after a third execution with any input value different from 5000 or 11000 is done (e.g., $(6500, \mathsf{true})$) the monitor would raise a flag indicating a violation of minimality occurred. Two important comments are needed here concerning a solution based on runtime monitoring: i) It might happen that the monitor never finds a witness for non-minimality even if the program is non-minimal. This could happen if the executions loop over the same inputs while never violating the property (at runtime a monitor can only act on the real executions of the program), or if the domain and co-domain of the program are too big, eventually needing an unbounded number of executions in order to be able to exactly produce the violating trace ii) It could happen that the program is non-minimal but there is a client-side minimiser filtering the inputs (for instance by choosing always a representative for each possible output). If this is the case then the monitor would never detect that the program is non-minimal. The latter case shows that what our monitoring approach is in fact doing is to check non-minimality for a *composed system* formed by the program (server side) and a potential minimiser (client side).

If we do have additional information about the input domain we can do better. In particular, if the input domain is finite for the monolithic case (and there is at most one infinite domain, for the distributed case), we can use runtime monitoring in a controlled environment in order to give a definitive Yes/No answer to the (non-)minimality problem. We combine the monitor as before but now we generate all possible inputs and check (exhaustively) all input/output pairs. Even better, we have a procedure that computes a minimiser for the given program. For our example, the minimiser is simply a program that generates two different constant numbers depending on whether the real input is less than 10000 (e.g., 5000) or bigger than 10000 (e.g., 15000). In this case, the program is still non-minimal, but we guarantee that it only receives two different values not disclosing the real input (the composed system

[2] The example is only for the monolithic case and is simple on purpose. In the rest of the paper we give a formalisation of all the concepts and present results for the more general case also (two versions of distributed minimality).

[3] For simplicity we consider here multiple executions of the program, that is, a set of traces of length one.

becomes minimal). Briefly, we can compute a minimiser for an arbitrary deterministic program under some reasonable assumptions (in practice most programs operate on bounded domains).

We have here summed-up some of the contributions of our paper and given an example for the simplest case (monolithic). In the rest of the paper we present our results in a more formal manner both for the monolithic and the more complex distributed case. We also give proof-of-concepts implementations for our monitoring approach.

More concretely, our contributions in this paper are:

1. *Monolithic case:* We formally define runtime monitors to check (non-)minimality for the monolithic case. We prove that checking minimality is not possible in general, but under certain conditions we can monitor both minimality and its negation (Section 3.2).
2. *Distributed case:* We show that distributed minimality and its negation are not monitorable in general. A strong version of distributed non-minimality is monitorable, and we formally define runtime monitor to check strong distributed (non-)minimality. Under certain conditions, we can monitor minimality and its negation for all cases (Section 4.2).
3. Based on the previous positive results, we give a procedure, using runtime monitoring in a controlled (pre-deployment) environment, that gives a definite answer on whether the program is minimal or not. (Section 5).
4. For our pre-deployment result (when the input vector contains at most one infinite input domain) we propose a procedure to generate a data minimiser (Section 5.2).
5. We provide proof-of-concept implementations for online runtime monitoring for the monolithic and distributed case (only for non-minimality), and for the pre-deployment monitoring for minimality and non-minimality for all cases. We also give an implementation for getting a minimiser for the monolithic case (Section 6).

2 Preliminaries

In this section we revisit basic concepts related to runtime verification (Section 2.1), and introduce other notations that we will use in the paper (Section 2.2).

According to [13] "*runtime verification* (RV) is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property". Checking whether an execution meets a correctness property is typically performed using a *monitor*, a program that decides whether the current execution satisfies the given property by outputting either yes/true or no/false. Formally, when $\|\varphi\|$ denotes the set of valid executions given by a property φ, RV boils down to checking whether a specific execution is an element of $\|\varphi\|$. Thus, in its mathematical essence, runtime verification answers the word problem, i.e. the problem whether a given word is included in some language. In general, the monitor is automatically extracted from the property, in this case showing that the monitor satisfies some desirable properties.

The system being monitored could be considered as a *black-box*, as for instance done by the RV approaches in [11, 2, 3, 12, 9], or as a *white-box* (or *grey-box*) as in [7]. Properties which the monitor should verify are usually specified in high-level formalisms with a semantics customised for finite executions such as automata theory [7, 9] or some variant of Linear Temporal Logic (LTL) such as LTL_3 [2].

A verification monitor does not influence or change the program execution. Such a monitor can be used to check the current execution of a system (*online*) or a stored execution of a system (*offline*). An execution of a system is considered as a finite sequence of actions emitted by a system being monitored. As illustrated in Figure 2, a verification monitor for a given property φ takes a sequence of events σ from a *black-box* system (event emitter) as input and produces a verdict as output that provides information about whether the current execution of the system σ satisfies φ or not.

In order to reason about runtime monitoring and verification, we rely on program execution traces. We thus need to introduce some additional basic notions before formally defining a runtime monitor.

A finite word over a finite alphabet Σ is a finite sequence $\sigma = a_1 \cdot a_2 \cdots a_n$ of elements of Σ. The set of all finite words over Σ is denoted by $\Sigma^{\#}$, and Σ^* denote a subset of $\Sigma^{\#}$ (i.e., $\Sigma^* \subseteq \Sigma^{\#}$). The *length* of a finite word σ is denoted by $|\sigma|$. The empty word over Σ is denoted by ϵ_Σ, or ϵ when clear from the context. The *concatenation* of two words σ and σ' is denoted as $\sigma \cdot \sigma'$. A word σ' is a *prefix* of a word

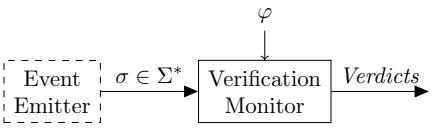


Fig. 2: Runtime Verification.

σ, denoted as $\sigma' \preccurlyeq \sigma$, whenever there exists a word σ'' such that $\sigma = \sigma' \cdot \sigma''$; and $\sigma' \prec \sigma$ if additionally $\sigma' \neq \sigma$; conversely σ is said to be an *extension* of σ'.

Given an n-tuple of symbols $e = (e_1, \ldots, e_n)$, for $i \in [1, n]$, $\Pi_i(e)$ is the projection of e on its i-th element, i.e., $\Pi_i(e) \stackrel{\text{def}}{=} e_i$. Given a word σ of length n, for any $i \in [1, n]$, σ_i denotes i^{th} element in σ.

2.1 Runtime verification monitor

In this section, we present a definition of a monitor for any given property φ, and present and discuss some important constraints that it satisfies.

A monitor is a device that reads/observes a finite trace (an execution of the system being monitored) and emits a verdict regarding satisfaction of a given property φ. The verdicts provided by the monitor belong to the set $\mathcal{D} = \{\top, \bot, ?\}$, where verdicts true ($\top$) and false ($\bot$) are conclusive verdicts while unknown (?) is an inconclusive verdict. A monitor for any given property φ is denoted as M_φ . Let us see a definition of a verification monitor for any given property $\varphi \subseteq \Sigma^*$.

Definition 1 (RV monitor). *Let $\sigma \in \Sigma^*$ denote a current observation of an execution of the system, and consider a property $\varphi \subseteq \Sigma^*$. A* monitor *is a function $M_\varphi : \Sigma^* \to \mathcal{D}$, where $\mathcal{D} = \{\top, \bot, ?\}$ defined as follows:*

$$M_\varphi(\sigma) = \begin{cases} \top & \text{if } \forall \sigma' \in \Sigma^* : \sigma \cdot \sigma' \in \varphi \\ \bot & \text{if } \forall \sigma' \in \Sigma^* : \sigma \cdot \sigma' \notin \varphi \\ ? & \text{otherwise} \end{cases}$$

Property φ is a set of finite words over alphabet Σ (i.e., $\varphi \subseteq \Sigma^*$). Verdicts *true* ($\top$) and *false* ($\bot$) are conclusive verdicts, and verdict *unknown* (?) is an inconclusive verdict.[4]

- $M_\varphi(\sigma)$ returns $\top$ if for any continuation $\sigma' \in \Sigma^*$, $\sigma \cdot \sigma'$ *satisfies* φ.
- $M_\varphi(\sigma)$ returns $\bot$ if for any continuation $\sigma' \in \Sigma^*$, $\sigma \cdot \sigma'$ *falsifies* φ.
- $M_\varphi(\sigma)$ returns unknown (?) otherwise.

Definition 2 (Monitorability). *Let $\varphi \subseteq \Sigma^*$ be a property. We say that φ is* monitorable *iff a monitor M_φ (as per Definition 1) exists for φ.*

Proposition 1. *For any given property $\varphi \subseteq \Sigma^*$ that is monitorable, monitor M_φ as per Definition 1 satisfies the following constraints:*
Impartiality $\forall \sigma \in \Sigma^*$,

$$\begin{array}{l} M_\varphi(\sigma) = ? \text{ iff} \\ (\sigma \in \varphi \wedge \exists \sigma' \in \Sigma^* : \sigma \cdot \sigma' \notin \varphi) \vee (\sigma \notin \varphi \wedge \exists \sigma' \in \Sigma^* : \sigma \cdot \sigma' \in \varphi) \end{array} \quad \textbf{(Imp)}$$

Anticipation $\forall \sigma \in \Sigma^*$,

$$\begin{array}{l} M_\varphi(\sigma) = \top \text{ iff } (\forall \sigma' \in \Sigma^* : \sigma \cdot \sigma' \in \varphi) \\ M_\varphi(\sigma) = \bot \text{ iff } (\forall \sigma' \in \Sigma^* : \sigma \cdot \sigma' \notin \varphi) \end{array} \quad \textbf{(Acp)}$$

Impartiality expresses that for a finite trace $\sigma \in \Sigma^*$, the monitor provides inconclusive verdict ? if and only if there exists a continuation of σ leading to another verdict. That is, if σ is consistent with φ, but there is some extension of σ which is not, or conversely, if σ is not consistent with φ but some extension is, then the monitor must give verdict ? on σ.

[4] Inconclusive verdict unknown (?), can be refined, where the monitor can provide information/verdict only about the execution seen so far.

Anticipation states that for a finite trace $\sigma \in \Sigma^*$, the monitor $M_\varphi(\sigma)$ should provide a conclusive verdict $\top$ (resp. $\bot$) iff every continuation of σ satisfies (resp. violates) φ. Thus, anticipation also means that if $M_\varphi(\sigma)$ is $\top$ (resp. $\bot$), then every continuation of σ also evaluates to $\top$ (resp. $\bot$). Formally for any $\sigma \in \Sigma^*$,

$$\begin{array}{c}(\text{ if } M_\varphi(\sigma) = \top \text{ then } (\forall \sigma' \in \Sigma^*, M_\varphi(\sigma \cdot \sigma') = \top) \textbf{ and} \\ (\text{ if } M_\varphi(\sigma) = \bot \text{ then } (\forall \sigma' \in \Sigma^*, M_\varphi(\sigma \cdot \sigma') = \bot)).\end{array}$$

Constraints **Imp** and **Acp** ensure that the monitor provides a conclusive verdict as soon as possible. That is, constraints **Imp** and **Acp** also ensure the following:

–

$$\begin{array}{r}\forall \sigma \in \Sigma^*, (M_\varphi(\sigma) = \bot \wedge \forall \sigma' \prec \sigma, M_\varphi(\sigma') =?) \\ \implies \forall \sigma' \prec \sigma, \exists \sigma'' \in \Sigma^* : \sigma' \cdot \sigma'' \in \varphi.\end{array}$$

–

$$\begin{array}{r}\forall \sigma \in \Sigma^*, (M_\varphi(\sigma) = \top \wedge \forall \sigma' \prec \sigma, M_\varphi(\sigma') =?) \\ \implies \forall \sigma' \prec \sigma, \exists \sigma'' \in \Sigma^* : \sigma' \cdot \sigma'' \notin \varphi.\end{array}$$

The terms impartiality and anticipation are introduced as requirements of monitors in other works related to runtime verification [12].

2.2 Programs and properties in the monolithic and distributed cases

We consider monitoring programs with deterministic behavior, where in every execution of the program, it consumes an input, and emits an output. Let I denote a finite set of inputs and O denote a finite set of outputs. The alphabet $\Sigma = I \times O$, and $\Sigma^\#$ is the set of finite words over Σ.

Remark 1. Since we focus on deterministic programs, for any given alphabet $\Sigma = I \times O$, we are interested in only finite words over alphabet Σ that do not contain input-output events which have the same input values but differ in their output values.

We consider $\Sigma^* \subseteq \Sigma^\#$, where $\forall \sigma \in \Sigma^*$, the following condition holds:

$$\forall i \in [1, |\sigma|], \forall j \in [1, |\sigma|], \text{ if } (\Pi_1(\sigma_i) = \Pi_1(\sigma_j) \text{ then } \Pi_2(\sigma_i) = \Pi_2(\sigma_j).$$

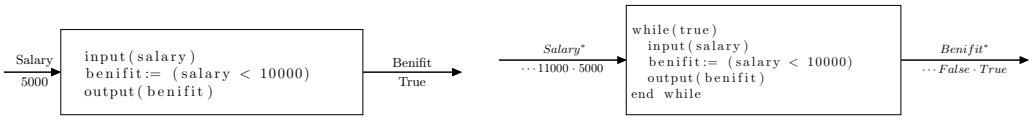
We consider monitoring both inputs and outputs of programs with deterministic behavior. A single execution of such a program is an input-output event $(i, o) \in I \times O$, where I denote a finite set of inputs, and O denotes a finite set of outputs.

We are interested in checking whether a program satisfies data minimality properties (introduced later in Sections 3.1, and 4.1). These properties can be modeled as hyper-properties [5, 10], where a hyper-property is a set of sets of traces. When we consider monitoring of hyperproperties, we need to consider multiple executions of the program being monitored, and analysis of sets of traces [4].

Consider the program to be some service provided by a web server. In practice, such a (service) functionality is not just used once, but multiple times with (different) inputs (e.g., different clients invoking the service). Thus, when we consider observing (monitoring) input-output behavior of such a program at runtime, we actually observe several executions of the program (i.e, can be considered as monitoring the program executed repeatedly in a loop).

We thus can consider the program in-loop as the program being monitored, and data minimality properties that we consider in this work can be formalized as normal trace properties, and the monitoring problem can be reduced to analysis of a single trace.

Remark 2 (Independence of events). Note that in the program in-loop each input-output computation is independent. The output produced by the program in each execution is only dependent on the input consumed in that particular execution.



(a) Example of a deterministic monolithic program $\mathcal{F}$

(b) Example of $\mathcal{P}$ (repeated execution of $\mathcal{F}$)

Fig. 3: Example of a monolithic program $\mathcal{F}$ and its associated program-in-loop $\mathcal{P}$

Monolithic case. In the monolithic case, the program has a single input source. We denote a deterministic program in the monolithic case as $\mathcal{F} : I \to O$, where I denote a finite set of inputs, and O denotes a finite set of outputs. The language of $\mathcal{F}$ is denoted as $\mathcal{L}(\mathcal{F})$, where $\mathcal{L}(\mathcal{F}) = \{(i, o) \in I \times O : o = \mathcal{F}(i)\}$, and $\mathcal{L}(\mathcal{F}) \subseteq I \times O$. Note that $\forall i \in I, \forall o \in O, (i, o) \in \mathcal{L}(\mathcal{F}) \implies (\forall o' \neq o \in O : (i, o') \notin \mathcal{L}(\mathcal{F}))$.

We consider monitoring both inputs and outputs of a program $\mathcal{F}$. A single execution of $\mathcal{F}$ is an input-output event $(i, o) \in I \times O$. We consider observing (monitoring) the input-output behavior over several executions of the program $\mathcal{F}$.

Let program $\mathcal{P}$ denote $\mathcal{F}$ executed repeatedly in a loop. An execution of $\mathcal{P}$ is an infinite sequence of input-output events $\sigma \in \Sigma^\omega$, where $\Sigma = I \times O$. The *behavior* of program $\mathcal{P}$ is denoted as $exec(\mathcal{P}) \subseteq \Sigma^\omega$. The *language* of $\mathcal{P}$ is denoted by $\mathcal{L}(\mathcal{P}) = \{\sigma \in \Sigma^* | \exists \sigma' \in exec(\mathcal{P}) \wedge \sigma \preccurlyeq \sigma'\}$ i.e. $\mathcal{L}(\mathcal{P})$ is the set of all finite prefixes of the sequences in $exec(\mathcal{P})$. Note that $\mathcal{L}(\mathcal{P})$ is prefix-closed, i.e., prefixes of any word that belongs to $\mathcal{L}(\mathcal{P})$ also belong to $\mathcal{L}(\mathcal{P})$.

Properties. A property φ over a finite alphabet Σ defines a set $\varphi \subseteq \Sigma^*$. A program $\mathcal{P} \models \varphi$ iff $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\varphi)$. Given a word $\sigma \in \Sigma^*$, $\sigma \models \varphi$ iff $\sigma \in \mathcal{L}(\varphi)$.

Example 1 (Deterministic monolithic program $\mathcal{F}$ and its corresponding program-in-loop $\mathcal{P}$). Let us consider a simple example illustrated in Figure 3. An example program $\mathcal{F} : I \to O$ is illustrated in Figure 3a, that takes salary information (which is an integer, i.e., set of possible inputs $I = \mathbb{N}$), and returns whether eligible for benefits or not (i.e., the set of possible outputs $O = \mathbb{B}$). The output of the program is true if salary is less than 10000, and false otherwise. When the program is fed with input 5000, it returns true as output.

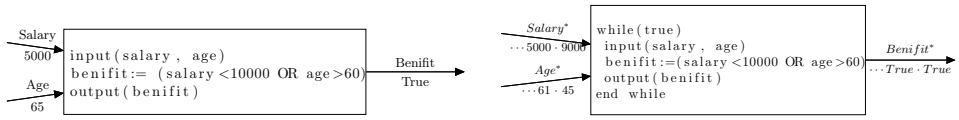
Figure 3b illustrates an example of program-in-loop $\mathcal{P}$ corresponding to repeated execution of $\mathcal{F}$ in Figure 3a. Thus, input to $\mathcal{P}$ is a stream of inputs events $\sigma_I \in I^*$, and the output is a stream of outputs $\sigma_O \in O^*$. In this example, $\sigma_I = 5000 \cdot 11000 \cdots$, and $\sigma_O = \mathsf{true} \cdot \mathsf{false} \cdots$. The set of input-output events $\Sigma = \mathbb{N} \times \mathbb{B}$, and a finite prefix of an execution of $\mathcal{P}$ is $(5000, \mathsf{true}) \cdot (11000, \mathsf{false})$, where in the first iteration of the while-loop, input is 5000 and output is true, and in the second iteration, input is 11000 and output is false. $(5000, \mathsf{true}) \in \mathcal{L}(\mathcal{P})$, and $(5000, \mathsf{true}) \cdot (11000, \mathsf{false}) \in \mathcal{L}(\mathcal{P})$.

Distributed case. In the distributed case, we consider that the deterministic program has more than one input sources. In every execution, it consumes an input from each of its source and it emits an output. We consider a finite number of input sources $n \geq 1$, where the set of input events $I = I_1 \times \cdots \times I_n$ where for all $i \in [1, n]$, I_i is a finite set of possible inputs for input source i, and an input event $(i_1, \cdots, i_n) \in I$, where $i_j \in I_j$. In every execution of the program, it consumes an input event $(i_1, \cdots, i_n) \in I$, and emits an output event $o \in O$, where O is a finite set of possible outputs. A deterministic program in the distributed case is denoted as $\mathcal{DF} : I_1 \times \cdots \times I_n \to O$.

Similar to the monolithic case, we consider monitoring inputs and outputs of $\mathcal{DF}$. Let program $\mathcal{DP}$ denote program $\mathcal{DF}$ executed repeatedly in a loop, and let $\Sigma = I \times O$ with $I = I_1 \times \cdots \times I_n$. An execution of $\mathcal{DP}$ is an infinite sequence of input-output events $\sigma \in \Sigma^\omega$. The *behavior* of program $\mathcal{DP}$ is denoted as $exec(\mathcal{DP}) \subseteq \Sigma^\omega$. The *language* of $\mathcal{DP}$ is denoted by $\mathcal{L}(\mathcal{DP}) = \{\sigma \in \Sigma^* | \exists \sigma' \in exec(\mathcal{DP}) \wedge \sigma \preccurlyeq \sigma'\}$ i.e. $\mathcal{L}(\mathcal{DP})$ is the set of all finite prefixes of the sequences in $exec(\mathcal{DP})$.

A property φ over Σ, where $\Sigma = I \times O$ with $I = I_1 \times I_2 \times \cdots \times I_n$, defines a set $\mathcal{L}(\varphi) \subseteq \Sigma^*$. A program $\mathcal{P} \models \varphi$ iff $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\varphi)$. Given a word $\sigma \in \Sigma^*$, $\sigma \models \varphi$ iff $\sigma \in \mathcal{L}(\varphi)$.

Example 2 (Program in the distributed case $\mathcal{DF}$, and its corresponding program -in-loop $\mathcal{DP}$). Let us consider a simple example illustrated in Figure 4. An example program $\mathcal{DF} : I_1 \times \cdots \times I_n \to O$ is illustrated in Figure 4a. In this example, the program has two input sources, salary which is an integer



(a) Example of a program $\mathcal{DF}$ in the distributed case $\mathcal{F}$

(b) Example of $\mathcal{DP}$ (repeated execution of $\mathcal{DF}$).

Fig. 4: Example of a program $\mathcal{DF}$ in the distributed case and its associated program-in-loop $\mathcal{DP}$

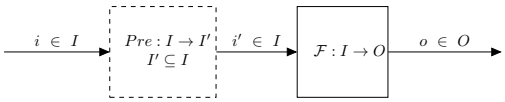


Fig. 5: Input data pre-processor (Monolithic case).

$I_1 = \mathbb{N}$, and age which is also an integer $I_2 = \mathbb{N}$. It checks eligibility for benefits depending on both salary and age information, and returns a Boolean as output i.e., $O = \mathbb{B}$. The output of the program is true if salary is less than 10000, or if age is greater than 60. When the program is fed with input 9000 for salary and 45 for age, it returns true as output.

Figure 4b illustrates an example of program $\mathcal{DP}$ which corresponds to repeated execution of $\mathcal{DF}$ in Figure 4a. Thus, input to $\mathcal{DP}$ is a stream of inputs events $\sigma_I \in I^*$ where $I = I_1 \times \cdots \times I_n$, and the output is a stream of outputs $\sigma_O \in O^*$. In this example, we have two input sources (salary and age), and example input word is $\sigma_I = (9000, 45) \cdot (5000, 61) \cdots$, and $\sigma_O = \mathsf{true} \cdot \mathsf{true} \cdots$. The set of input-output events $\Sigma = I \times O$, where $I = \mathbb{N} \times \mathbb{N}$ and $O = \mathbb{B}$, and a finite prefix of an execution of $\mathcal{DP}$ is $((9000, 45), \mathsf{true}) \cdot ((5000, 61), \mathsf{true})$, where in the first iteration of the while-loop, input is $(9000, 45)$ and output is true, and in the second iteration, input is $(5000, 61)$ and output is true. $((9000, 45), \mathsf{true}) \in \mathcal{L}(\mathcal{DP})$, and $((9000, 45), \mathsf{true}) \cdot ((5000, 61), \mathsf{true}) \in \mathcal{L}(\mathcal{DP})$.

3 Monolithic case: Data minimality and detection of (non) minimality via monitoring

In this section, we will focus on the runtime monitoring framework for the monolithic case. We first recall and formally introduce the data minimality principle[5], and we introduce the notion of *non-minimality* (Section 3.1). Later, we define a monitoring mechanism to detect minimality (resp. non-minimality) by observing input-output behavior of a program (Section 3.2). We show that non-minimality property is monitorable, and minimality property is not monitorable in general. Minimality is also monitorable when the input domain of the program being monitored is bounded and the monitor has knowledge about the input domain.

3.1 Data minimality in the monolithic case

We first introduce the data minimality principle in the monolithic case. Data minimality ensures that the range of inputs provided to a program is reduced such that when two inputs result in the same response, then one of them can be considered redundant. Ideally, a program satisfying data minimization principle should be one such that the cardinality of the output domain is equal to the cardinality of the input domain.

In Definition 4, as illustrated in Figure 5, we assume that there is a data pre-processor which is a function Pre from I to I that transforms inputs before they are fed to an *un-trusted* program $\mathcal{F} : I \to O$.

Definition 3 (Pre-processor). *Given a program $\mathcal{F} : I \to O$, we say that $Pre : I \to I$ is a pre-processor for $\mathcal{F}$ iff:*

[5] Our presentation of minimisation here is slightly different from the one given in [1]. The main differences are that we define data minimality as a derived concept from strong dependency instead of a characterization from the definition of minimisers.

```
inputPre(salary)
  if (salary < 6000)
    salaryRep = 1000
  else if  (6000 < salary <10000)
    salaryRep = 6000
  else:
    salaryRep= 10000
return(salaryRep)
```

Fig. 6: Example of an input pre-processor for program $\mathcal{F}$ illustrated in Figure3a.

1. $\forall i \in I : \mathcal{F}(Pre(i)) = \mathcal{F}(i)$.
2. $\forall i \in I : Pre(i) = Pre(Pre(i))$.

Condition 1 states that the pre-processor should not change the behavior of the program. For any input $i \in I$, the output that the program produces by consuming the pre-processed input should be equal to the output it produces by directly consuming the input i.

Condition 2 states that for any input $i \in I$, if we feed the pre-processed input to the pre-processor again, then it returns back the same pre-processed input.

Pre-processors perform some degree of domain reduction, and range(Pre) $\subseteq I$ (with range(Pre) denoting the range of function Pre, i.e., $\{Pre(i)|i \in I\}$. In case there is no pre-processor, in theory it could be considered that there is a pre-processor that is the identity function.

Example 3 (Input data pre-processor). Figure 6 presents an example input pre-processor for the program $\mathcal{F}$ illustrated in Figure3a. If salary is less than 6000 then it is mapped to representative 1000, if salary is greater than 6000 and less than 10000 it is mapped to 6000, and it is mapped to 10000 otherwise.

Definition 4 expresses that a program $\mathcal{F} : I \rightarrow O$, where I is the set of possible inputs, $I' \subseteq I$ and O is the set of possible outputs, is monolithic minimal for I' if for any two inputs $i_1, i_2 \in I'$, where i_1 is different from i_2, the output that program $\mathcal{F}$ produces for input i_2 should differ from the output that it produces for input i_1.

Definition 4 (Monolithic minimality of program $\mathcal{F}$). *A program $\mathcal{F} : I \rightarrow O$ is monolithic minimal for $I' \subseteq I$* iff *the following condition holds:*

$$\forall i_1, i_2 \in I',\ \textit{if } i_1 \neq i_2 \textit{ then } \mathcal{F}(i_1) \neq \mathcal{F}(i_2).$$

Remark 3. Note that when we say that $\mathcal{F}$ is minimal (non-minimal), we mean that the composition of $\mathcal{F}$ with a pre-processor ($Pre : I \rightarrow I'$) is minimal (non-minimal). We consider the scenario illustrated in Figure 6. If composition of $\mathcal{F}$ with Pre is minimal (non-minimal), and if Pre is the identity function then $\mathcal{F}$ itself is minimal (non-minimal).

We now introduce the notion of *non-minimality* of program $\mathcal{F}$, which is obtained in a straightforward manner by negating the constraint for minimality of program $\mathcal{F}$ in Definition 4. When a given deterministic program $\mathcal{F}$ satisfies this non-minimality property, then the input given to this program are not minimized in the best possible manner.

Definition 5 expresses that a given program $\mathcal{F} : I \rightarrow O$, where I is the set of possible inputs, $I' \subseteq I$ and O is the set of possible outputs, $\mathcal{F}$ is monolithic non-minimal for I' if there exists two inputs $i_1, i_2 \in I'$, where i_1 is different from i_2, and the program $\mathcal{F}$ produces the same output for inputs i_1 and i_2.

Definition 5 (Monolithic non-minimality of program $\mathcal{F}$). *A program $\mathcal{F} : I \rightarrow O$ is non-minimal for $I' \subseteq I$ iff the following condition holds:*

$$\exists i_1, i_2 \in I' : i_1 \neq i_2 \wedge \mathcal{F}(i_1) = \mathcal{F}(i_2).$$

Example 4 (Monolithic non-minimality). Let us consider the program presented in Figure 3a as $\mathcal{F}$, and the function presented in Figure 6 as the input data pre-processor. Thus, I' in this example is $\{1000, 6000, 10000\}$. Program $\mathcal{F}$ is monolithic non-minimal w.r.t I' since there are two elements 1000 and 6000 in I', and $\mathcal{F}(1000) = \mathcal{F}(6000) = \mathsf{true}$.

We are interested in defining minimality (resp. non-minimality) as trace properties. As discussed in preliminaries, let $\mathcal{P}$ denote a program that executes program $\mathcal{F} : I \to O$ repeatedly. The language of $\mathcal{P}$ is $\mathcal{L}(\mathcal{P}) \subseteq \Sigma^*$, where $\Sigma = I \times O$ (see Section 2).

Monolithic minimality property $\varphi_m \subseteq \Sigma^*$, is the set of all words in Σ^*, such that for any word $\sigma \in \varphi_m$, for any two events at different indexes in σ, if the projection on inputs of the two events differ, then the projection on outputs of the two events should also differ. Property φ_m is formally defined as follows:

Definition 6 (Monolithic minimality property φ_m). *Given alphabet Σ, where $\Sigma = I \times O$, property $\varphi_m \subseteq \Sigma^*$, is the set of all words belonging to Σ^* satisfying the following constraint:*

$$\begin{aligned}&\forall \sigma \in \varphi_m,\\ &\quad \forall i \in [1, |\sigma|], \forall j \in [1, |\sigma|],\\ &\qquad \textit{if } (\Pi_1(\sigma_i) \neq \Pi_1(\sigma_j)) \textit{ then } (\Pi_2(\sigma_i) \neq \Pi_2(\sigma_j))\end{aligned}$$

Remark 4 (φ_m is prefix-closed). Note that all prefixes of all word belonging to φ_m also belong to φ_m, that is, property φ_m is prefix-closed.

A prefix of an execution of a program $\sigma \in \mathcal{L}(\mathcal{P})$ where $\mathcal{L}(\mathcal{P}) \subseteq \Sigma^*$ satisfies property φ_m iff $\sigma \in \varphi_m$.

Example 5 (Monolithic minimality property). Consider program $\mathcal{P}$ to be the example program illustrated in Figure 3b. Consider a prefix of an execution of this program $\sigma_1 = (5000, \mathsf{true}) \cdot (11000, \mathsf{false})$ which belongs to $\mathcal{L}(\mathcal{P})$. $\Sigma_1 \in \varphi_m$. Consider another prefix of an execution of this program $\sigma_2 = (5000, \mathsf{true}) \cdot (11000, \mathsf{false}) \cdot (8000, \mathsf{true})$ where $\sigma_2 \in \mathcal{L}(\mathcal{P})$. Note that $\sigma_2 \notin \varphi_m$ since if we consider input-output events at index 1 and index 3, input values in these two events differ (5000 and 8000) but the output values are equal (true in both the events).

We now define monolithic non-minimality property, which is negation of the minimality property φ_m introduced in Definition 6.

Definition 7 (Monolithic non-minimality property $\overline{\varphi_m}$). *Given alphabet Σ where $\Sigma = I \times O$, property $\overline{\varphi_m}$ is the set of all words in Σ^* satisfying the following constraint:*

$$\begin{aligned}&\forall \sigma \in \overline{\varphi_m} :\\ &\quad \exists i \in [1, |\sigma|], \exists j \in [1, |\sigma|] :\\ &\qquad (\Pi_1(\sigma_i) \neq \Pi_1(\sigma_j) \wedge \Pi_2(\sigma_i) = \Pi_2(\sigma_j))\end{aligned}$$

Remark 5 (Property $\overline{\varphi}_m$ is extension-closed). Note, that property $\overline{\varphi}_m$ is extension closed, i.e., for any word σ that belongs to φ_m, every possible extension of σ also belongs to $\overline{\varphi}_m$. Formally, $\forall \sigma \in \Sigma^* : \sigma \in \overline{\varphi}_m \implies (\forall \sigma' \in \Sigma^* : \sigma \preccurlyeq \sigma' \implies \sigma' \in \overline{\varphi}_m)$.

Example 6. Consider the example program $\mathcal{P}$ illustrated in Figure 3b. Consider a prefix of an execution of this program $\sigma_1 = (5000, \mathsf{true}) \cdot (11000, \mathsf{false})$ which belongs to $\mathcal{L}(\mathcal{P})$. $\sigma_1 \notin \overline{\varphi}_m$. Consider another prefix of an execution of this program $\sigma_2 = (5000, \mathsf{true}) \cdot (11000, \mathsf{false}) \cdot (8000, \mathsf{true})$ where $\sigma_2 \in \mathcal{L}(\mathcal{P})$. Note that $\sigma_2 \in \overline{\varphi}_m$ since if we consider input-output events at index 1 and index 3, input values in these two events differ (5000 and 8000) but the output values are equal (true in both the events). Any possible continuation of σ_2 also belong to $\overline{\varphi}_m$.

Lemma 1 ($\overline{\varphi_m} = \Sigma^* \setminus \varphi_m$). *Note that $\overline{\varphi_m}$ is the negation of property φ_m, that is, $\overline{\varphi_m} = \Sigma^* \setminus \varphi_m$. A word $\sigma \in \Sigma^*$ satisfies $\overline{\varphi_m}$ if $\sigma \in \overline{\varphi_m}$. It follows that:*

- $\forall \sigma \in \Sigma^*, \sigma \in \varphi_m \implies \sigma \notin \overline{\varphi}_m$;
- $\forall \sigma \in \Sigma^*, \sigma \in \overline{\varphi}_m \implies \sigma \notin \varphi_m$.

Lemma 1 is immediate consequence of Definitions 6 and 7.

Let us consider $\mathcal{F}$ and its corresponding program-in-loop $\mathcal{P}$. The following theorem states that, if there exists an observation of an execution of program $\mathcal{P}$ that satisfies the non-minimality property, then function $\mathcal{F}$ is non-minimal. If every word that belongs to $\mathcal{L}(\mathcal{P})$ also belongs to property φ_m (i.e., every possible observation of execution of $\mathcal{P}$ satisfies φ_m), then $\mathcal{F}$ is monolithic minimal.

Theorem 1. *Given $\mathcal{F} : I \to O$, let $\mathcal{P} \subseteq \Sigma^*$ where $\Sigma = I \times O$ correspond to the program-in-loop for $\mathcal{F}$, the following properties hold:*

- *$\mathcal{F}$ is monolithic non-minimal iff $\exists \sigma \in \Sigma^* : \sigma \in \mathcal{L}(\mathcal{P}) \wedge \sigma \in \overline{\varphi}_m$;*
- *$\mathcal{F}$ is monolithic minimal iff $\forall \sigma \in \Sigma^* : \sigma \in \mathcal{L}(\mathcal{P}) \implies \sigma \in \varphi_m$.*

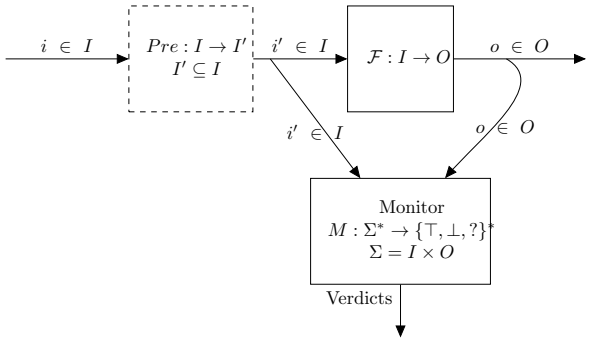


Fig. 7: Problem overview: monolithic case.

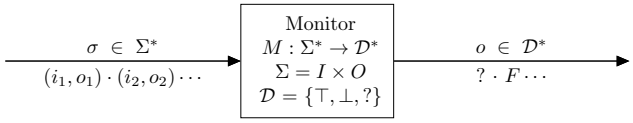


Fig. 8: Monitor M.

3.2 Monitoring mechanism to detect (non) minimality

We consider monitoring input-output behavior of program $\mathcal{P}$, where $\mathcal{P}$ is repeated execution of program $\mathcal{F} : I \to O$. Inputs to program $\mathcal{F}$ may be fed by a user (which can be considered as monitoring after deployment), or by a *testInputGenerator* which can be considered as testing $\mathcal{F}$ for data minimality in a controlled environment prior to deployment.

By monitoring $\mathcal{P}$ (input-output behavior of several executions of program $\mathcal{F}$), we are interested in checking whether an execution of $\mathcal{P}$ satisfies (resp. violates) minimality property φ_m.

The general context of the proposed monitoring approach is depicted in Figure 7. The inputs that the user (or *testInputGenerator*) provides belong to the set I. We consider that inputs from the user may be first pre-processed by a data pre-processor (Pre), and the pre-processed input that belongs to the set I' (where $I' \subseteq I$) is fed as input to the *untrusted* program $\mathcal{F}$. For each execution of $\mathcal{F}$, the monitor observes both the pre-processed input and the output of $\mathcal{F}$.

Note that we also assume that the monitor cannot observe and is not aware of the actual inputs provided by a user. Moreover, a pre-processor may not exist (i.e., can be considered as the identity function). We also assume that the monitor is unaware whether a pre-processor exists or not, and also does not know about its behavior and the set of all possible outputs I' of the pre-processor. The monitor observes the pre-processed input i' at runtime, that belongs to I' which also belongs to I.

Let us recall that the set of input-output events that the monitor can observe (receive) as input is denoted using Σ, where $\Sigma = I \times O$, and an input-output event is denoted as (i, o) where $i \in I$ and $o \in O$. After n executions of the program $\mathcal{F}$, the monitor observes a word $\sigma = (i_1, o_1), \cdots, (i_n, o_n) \in \mathcal{L}(\mathcal{P})$ as input. This is illustrated in Figure 8.

After each execution of $\mathcal{F} : I \to O$ (i.e., in every iteration of program $\mathcal{P}$), the monitor observes the (pre-processed) input and the output of program $\mathcal{F}$ in that particular iteration (step) of $\mathcal{P}$.

For any word $\sigma \in \Sigma^*$ (current observation of execution of $\mathcal{P}$) where $|\sigma| > 1$, M_{φ_m} as per Definition 1 is a monitor for property φ_m. The monitor returns `true` ($\top$) when σ followed by any extension of it satisfies the minimality property φ_m. The monitor returns $false$ ($\bot$) when the current observation of execution of $\mathcal{P}$ followed by any extension of it violates φ_m (resp. satisfies $\overline{\varphi}_m$). It returns ? (unknown) for the current observation if the other two cases do not hold.

We do not define properties in a finite representation (such as automata) or using some logic such as LTL, and moreover properties we consider are not related to checking constraints on the order in which certain actions should happen. Our properties φ_m (resp. $\overline{\varphi_m}$) are related to checking some relation between the input-output values.

We thus reduce checking whether the property is satisfied (resp. violated) for every extension of the current observation, to checking whether the current observation satisfies (resp. violates) some conditions.

We now introduce function sat_{φ_m} that is defined based on definitions of properties φ_m (resp. $\overline{\varphi_m}$). This function is used to check whether the current observation σ satisfies property φ_m (resp. $\overline{\varphi_m}$) .

Function $\mathsf{sat}_{\varphi_m} : \Sigma^* \to \mathbb{B}$, takes an input-output word and returns a Boolean as output. It is defined as follows:

$$\mathsf{sat}_{\varphi_m}(\sigma) = \begin{cases} \mathsf{true} & \text{if } \begin{array}{l} \forall i \in [1, |\sigma|], \forall j \neq i \in [1, |\sigma|] : \\ \qquad \Pi_1(\sigma_i) \neq \Pi_1(\sigma_j) \implies \Pi_2(\sigma_i) \neq \Pi_2(\sigma_j) \end{array} \\ \mathsf{false} & \mathit{Otherwise} \end{cases}$$

For any given word σ, $\mathsf{sat}_{\varphi_m}(\sigma)$ returns true if $\sigma \in \varphi_m$, and returns false otherwise, i.e, if it returns false, then $\sigma \in \overline{\varphi_m}$.

Note that from the definition of $\overline{\varphi_m}$, if $\sigma \in \overline{\varphi_m}$, then $\exists i \in [1, |\sigma|], \exists j \neq i \in [1, |\sigma|] : (\Pi_1(\sigma_i) \neq \Pi_1(\sigma_j) \wedge \Pi_2(\sigma_i) = \Pi_2(\sigma_j))$.

The following proposition expresses that if a word σ belongs to the property $\overline{\varphi_m}$, then every possible extension of it also belongs to the property $\overline{\varphi_m}$.

Proposition 2. $\forall \sigma \in \Sigma^*$, *if* $\sigma \in \overline{\varphi_m}$ *then* $(\forall \sigma' \in \Sigma^* : \sigma \preccurlyeq \sigma' \implies \sigma' \in \overline{\varphi_m})$.

Example 7. Consider the example program $\mathcal{P}$ illustrated in Figure 3b. Consider a prefix of an execution of this program $\sigma_1 = (5000, \mathsf{true}) \cdot (11000, \mathsf{false})$ which belongs to $\mathcal{L}(\mathcal{P})$. We have $\mathsf{sat}_{\varphi_m}(\sigma_1) = \mathsf{true}$. Consider another prefix of an execution of this program $\sigma_2 = (5000, \mathsf{true}) \cdot (11000, \mathsf{false}) \cdot (8000, \mathsf{true})$ where $\sigma_2 \in \mathcal{L}(\mathcal{P})$. We have $\mathsf{sat}_{\varphi_m}(\sigma_2) = \mathsf{false}$. For any word $\sigma' \in \Sigma^*$, $\mathsf{sat}_{\varphi_m}(\sigma_2 \cdot \sigma')$ will be false (i.e., $\sigma_2 \cdot \sigma' \in \overline{\varphi_m}$).

Remark 6 (Condition for conclusive verdict $\bot$). From Proposition 2, we can reduce the condition of the second case in M_{φ_m} as per Definition 1 for property φ_m to checking whether the current observation σ satisfies $\overline{\varphi}_m$.

Remark 7 (Impossibility of checking condition of the first case (satisfaction of property φ_m)). Note that regarding the condition of the $\top$ case (satisfaction of φ_m), checking whether the current observed word σ belongs to φ_m (i.e.,whether $\mathsf{sat}_{\varphi_m}(\sigma)$ is true) is not sufficient, and does not ensure that every extension of σ will also belong to φ_m if σ belongs to φ_m. Thus, testing condition of the first case is not possible in general.

By providing the monitor with knowledge about the input domain and when the input domain is bounded, it is possible to test the condition of the first case related to satisfaction of property φ_m.

We also introduce function in-ex : $I \times \Sigma^* \to \mathbb{B}$ where $\Sigma = I \times O$, that is used to test whether every input belonging to the set I appear at least once in a given input-output word $\sigma \in \Sigma^*$. It is defined as follows:

$$\text{in-ex}(I, \sigma) = \begin{cases} \mathsf{true} & \text{if } (\forall i \in I, \exists id \in [1, |\sigma|] : \Pi_1(\sigma_{id}) = i) \\ \mathsf{false} & \mathit{Otherwise} \end{cases}$$

Remark 8 (Condition for conclusive verdict $\top$ (satisfaction of property φ_m)). Note that the condition of the first case in M_{φ_m} as per Definition 1 for property φ_m reduces to checking whether σ satisfies the following two conditions:

- every input belonging the set of inputs I appear at least once in σ, i.e., $\text{in-ex}(I, \sigma) = \mathsf{true}$ and
- σ satisfies φ_m, i.e., $\mathsf{sat}_{\varphi_m}(\sigma) = \mathsf{true}$.

Note that if σ contains every input belonging to the set of inputs I, and if σ satisfies φ_m, then every possible extension of σ also satisfies φ_m.

Proposition 3. *Given any word $\sigma \in \Sigma^*$, where $\Sigma = I \times O$, and $|\sigma| > 1$, we have;*

$$\textit{if } (\text{in-ex}(I, \sigma) \wedge \mathsf{sat}_{\varphi_m}(\sigma)) \textit{ then } (\forall \sigma', \textit{ if } \sigma \preccurlyeq \sigma' \textit{ then } \sigma' \in \varphi_m).$$

Thus, using Propositions 2, 3, the conditions of the first two cases in M_{φ_m} can be simplified. We present the alternative simplified definition below, where the conditions of the first two cases are reduced to checking whether the observed input word satisfies some conditions.

Consider property $\varphi_m \subseteq \Sigma^*$ where $\Sigma = I \times O$. Let $\sigma \in \Sigma^*$ denote a finite input-output word over the alphabet $\Sigma = I \times O$. A monitor for property φ_m (resp. $\overline{\varphi_m}$) is a function $M_{\varphi_m} : \Sigma^* \to \mathcal{D}$, where $D = \{\top, \bot, ?\}$. For $\sigma = \epsilon$ and any word σ of length 1, $M(\sigma) =?$. Monitor M_{φ_m} is defined as follows:

σ	$M(\sigma)$
$(5000, \mathsf{true})$	?
$(5000, \mathsf{true}) \cdot (11000, \mathsf{false})$	?
$\mathbf{(5000, true)} \cdot (11000, \mathsf{false}) \cdot \mathbf{(8000, true)}$	$\bot$
$\mathbf{(5000, true)} \cdot (11000, \mathsf{false}) \cdot \mathbf{(8000, true)} \cdot (12000, \mathsf{false})$	$\bot$
$\mathbf{(5000, true)} \cdot (11000, \mathsf{false}) \cdot \mathbf{(8000, true)} \cdot (12000, \mathsf{false}) \cdots$	$\bot$

Table 1: Example illustrating behavior of the monitor M_{φ_m}.

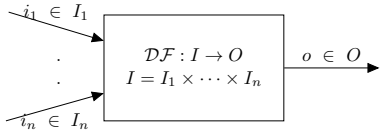


Fig. 9: Program in the distributed case with multiple inputs.

Definition 8 (Monitor M_{φ_m}). *A monitor for property φ_m (resp. $\overline{\varphi_m}$) is a function $M_{\varphi_m} : \Sigma^* \to \mathcal{D}$, where $D = \{\top, \bot, ?\}$ is defined as follows:*

$$M_{\varphi_m}(\sigma) = \begin{cases} \top & \textit{if } \ |\sigma| > 1 \wedge \text{in-ex}(I, \sigma) \wedge \mathsf{sat}_{\varphi_m}(\sigma) \\ \bot & \textit{if } \ |\sigma| > 1 \wedge \neg\mathsf{sat}_{\varphi_m}(\sigma) \\ ? & \textit{Otherwise} \end{cases}$$

Proposition 4. *M_{φ_m} in Definition 8 is a monitor for property φ_m as per Definition 1 (i.e., M_{φ_m} is sound and optimal).*

Example 8 (Example illustrating behavior of the monitor M_{φ_m}).
Let us again consider the example program $\mathcal{P}$ illustrated in Figure 3b. In Table 1, we present some example observations of an execution of program $\mathcal{P}$ being monitored denoted as σ, and the verdict provided by the monitor for σ. Initially, when the first event observed in $(5000, \mathsf{true})$, the monitor returns verdict unknown (?). In each step current observation is extended with a new event. Let the new event observed in the second step be $(11000, \mathsf{false})$. For current observation $\sigma = (5000, \mathsf{true}) \cdot (11000, \mathsf{false})$, the monitor returns verdict unknown. After observing the third event $(8000, \mathsf{false})$, the monitor returns verdict false ($\bot$) for $\sigma = (5000, \mathsf{true}) \cdot (11000, \mathsf{false}) \cdot (8000, \mathsf{false})$.

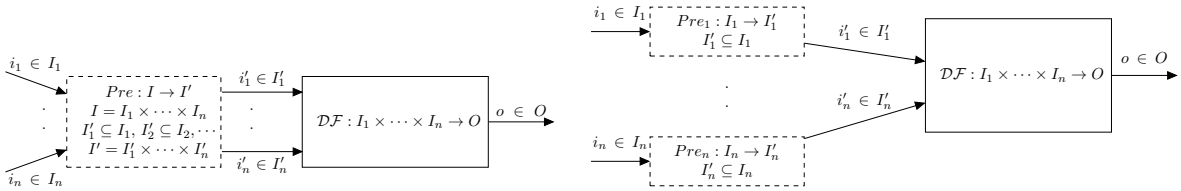
Remark 9 (Monitor with two cases). Note that when it is not possible to test whether current observation σ covers all inputs (i.e., when the input domain I is unknown), it is not possible to compute in-ex(I, σ). In this case, one can consider monitor with two cases (where the first case is merged with the unknown case). The monitor returns $\bot$ indicating violation of minimality (resp. satisfaction of non-minimality), if $\neg\mathsf{sat}_{\varphi_m}(\sigma)$ and ? otherwise.

4 Distributed case: Data minimality and detection of (non) minimality via monitoring

In Section 3, we considered that the program has a single input source. However, in several domains such as web services, a service provider requires data from multiple sources.

In this section, we introduce data minimality (and non-minimality) policies for the distributed case (Section 4.1). Distributed minimality (resp. distributed non-minimality) are not monitorable in general. Results we obtained for minimality (resp. non-minimality) in the monolithic case can be extended for *strong* distributed minimality (resp. *strong* distributed non-minimality). We present a monitoring mechanism to detect strong distributed minimality (resp. non-minimality) by observing input-output behavior of a program (Section 4.2).

A program in the distributed case with multiple input sources is illustrated in Figure 9. We consider deterministic programs that can be considered as functions with multiple input sources (e.g., multiple



(a) Single pre-processor (b) Multiple pre-processors (one per input source))

Fig. 10: Input data pre-processor (Distributed case).

clients). In each execution of the program, it consumes an input event from all its input sources and it emits an output event.

As discussed in Section 2, a program in the distributed case with n input sources can be considered as a function, denoted as $\mathcal{DF} : I_1 \times \cdots \times I_n \to O$. We consider monitoring input-output behavior of multiple executions of program $\mathcal{DF}$. Repeated execution of $\mathcal{DF}$ is denoted as $\mathcal{DP}$, where $\mathcal{DP} : I^* \to O^*$, with $I = I_1 \times \cdots \times I_n$. In Figure 4, we present an example of a program $\mathcal{DF}$ with two input sources and its corresponding program $\mathcal{DP}$.

Monolithic minimality will be too restrictive in the distributed case where we have multiple input sources. Let us consider the following examples taken from [1]. Let $\mathcal{DF}$ be the program $\text{XOR} : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ that takes two Boolean inputs and returns a Boolean as output. Since $\text{XOR}(0,0) = \text{XOR}(1,1)$, it follows that XOR is not monolithic minimal. The program $\text{OR} : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ is also not monolithic minimal since $\text{OR}(0,1) = \text{OR}(1,0)$. Monolithic minimality is in general not suitable (too strong a notion) for programs with multiple input sources.

Let us first understand the notions of distributed minimality, where the program being monitored has multiple input sources.

4.1 Data minimality in the distributed setting

In the following definitions, similar to the monolithic case, as illustrated in Figure 10a, we assume that there may be a pre-processor which can be considered as a function $Pre : I_1 \times \cdots \times I_n \to I'_1 \times \cdots \times I'_n$ (where for all $i \in [1,n]$, $I'_i \subseteq I_i$) that transforms user inputs before they are fed to the program. As illustrated in Figure 10b, there can be multiple pre-processors one for each input source. Pre-processor(s) may not exist, i.e., they can be considered as identity function(s) (forwarding user inputs to the program).

Definition 3 of a pre-processor in the monolithic case can be extended to the distributed case, and we omit details here (See [1] for details and definitions[6]).

We first present the notion of distributed minimality considered in [1], and introduce distributed non-minimality.

Definition 9 (Distributed minimality of program $\mathcal{DF}$). *Program $\mathcal{DF} : I_1 \times \cdots \times I_n \to O$ is* distributed minimal *for $I' \subseteq I$ iff for every input stream I_{id} where $id \in \{1, \cdots, n\}$, for any two different values $u, v \in I_{id}$, there are at least two input events $(i_1, \cdots, i_n), (i'_1, \cdots, i'_n) \in I$ which differ in exactly one input stream value (where $i_{id} = u$ and $i'_{id} = v$), and the program $\mathcal{DF}$ produces different output for $(i_1, \cdots, i_n)$ and $(i'_1, \cdots, i'_n)$. Formally,*

$$\begin{aligned}
&\forall id \in [1,n], \forall u, v \in I_{id} \text{ such that } u \neq v, \\
&\quad \exists i_1, i_2 \in I' : ((\Pi_{id}(i_1) = u \wedge \Pi_{id}(i_2) = v) \\
&\qquad \wedge (\forall j \in [1,n] : j \neq id \implies \Pi_j(i_1) = \Pi_j(i_2)) \\
&\qquad\quad \wedge \mathcal{DF}(i_1) \neq \mathcal{DF}(i_2))
\end{aligned}$$

Example 9. Distributed minimality is a weakening of the monolithic minimality (Definition 4). For example, the $\text{OR} : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ function, which was shown not to be monolithic minimal, is distributed-minimal. We have two input sources $I_1 \times I_2$ ($\mathbb{B} \times \mathbb{B}$). For the first input source, for each possible pair of distinct values in that position (that is $(0, \_)$ and $(1, \_)$), we can find satisfactory input tuples yielding different results (e.g., $((0,0),(1,0))$ since $\text{OR}(0,0) \neq \text{OR}(1,0)$).[7] Similarly, for input source 2, for each possible

[6] In [1], pre-processors are called *minimisers*, and minimisers are defined as *best minimisers*.

[7] Note that the pair $((0,1),(1,1))$ would not satisfy the definition, but this is fine as the definition only requires that at least one such tuple exists.

pair of distinct values in that position ($(\_, 0)$ and $(\_, 1)$), we have that the tuples $(0, 0)$ and $(0, 1)$ satisfy the definition ($\text{OR}(0, 0) \neq \text{OR}(1, 0)$).

We now introduce distributed non-minimality of a program $\mathcal{DF}$, as a negation of Definition 9.

Definition 10 (Distributed non-minimality). *Program $\mathcal{DF} : I_1 \times \cdots \times I_n \to O$ is* distributed non-minimal *for $I' \subseteq I$ iff there is an input stream I_{id} where $id \in [1, n]$, such that there exist two different values $u, v \in I_{id}$ such that for any two input events i_1 and i_2 that belong to I' where the value corresponding to input source id is v is one and u in the other, and the values of other input sources are equal in both i_1 and i_2, the program produces the same output for i_1 and i_2. Formally,*

$$\begin{aligned}&\exists id \in [1, n], \exists u, v \in I_{id} \text{ such that } u \neq v,\\ &\quad \forall i_1, i_2 \in I' : ((\Pi_{id}(i_1) = u \wedge \Pi_{id}(i_2) = v) \wedge\\ &\qquad (\forall j \in [1, n] : j \neq id \implies \Pi_j(i_1) = \Pi_j(i_2)))\\ &\qquad\quad \implies \mathcal{DF}(i_1) = \mathcal{DF}(i_2)\end{aligned}$$

Remark 10 (Non-monitoribality of distributed minimality (resp. non-minimality)). Both satisfaction and violation of distributed minimality are not monitorable in general. When monitoring a program $\mathcal{DP}$ (repeated execution of program $\mathcal{DF}$), detection of violation of distributed minimality also requires all the input domains to be known and bounded, and checking whether the current observation of execution of $\mathcal{DP}$ covers all the inputs.

We thus consider a variant of distributed minimality, called as *strong* distributed minimality. Later in Section 4.2, we show that the results related to monitoring for minimality (respectively non-minimality) in the monolithic case can be extended to monitoring for strong distributed minimality (respectively strong distributed non-minimality) in the distributed case.

Definition 11 (Strong distributed minimality of program $\mathcal{DF}$). *Program $\mathcal{DF} : I_1 \times \cdots \times I_n \to O$, where for all input sources $id \in [1, n]$, I_{id} is the set of possible inputs from source id, $I' \subseteq I$, and O is the set of possible outputs, $\mathcal{DF}$ is* strongly distributed minimal *for I' iff:*
for any two input events $(i_1, \cdots, i_n)$ and $(i'_1, \cdots, i'_n)$ belonging to I' that differ exactly in one element, the output that the program $\mathcal{DF}$ produces for input $(i'_1, \cdots, i'_n)$ is different from the output that it produces for input $(i_1, \cdots, i_n)$. Formally,

$$\begin{aligned}&\forall (i_1, \cdots, i_n), (i'_1, \cdots, i'_n) \in I' :\\ &\quad (\exists j \in [1, n] : i_j \neq i'_j \wedge \forall k \in [1, n] : k \neq j \implies i_k = i'_k) \implies\\ &\qquad \mathcal{DF}((i_1, \cdots, i_n)) \neq \mathcal{DF}((i'_1, \cdots, i'_n))\end{aligned}$$

Remark 11. When the number of input sources is one, strong distributed minimality (Definition 11) reduces to the monolithic minimality (Definition 4).

Example 10. Strong distributed minimality is also a weakening of the monolithic minimality (Definition 4). For example, we already saw that the $\text{XOR} : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ function is not monolithic minimal, since $\text{XOR}(0, 0) = \text{XOR}(1, 1)$. We can easily notice that the XOR function is strong distributed minimal since the output differs for every possible pair of input tuples differing exactly at one position (e.g., $\text{XOR}(0, 0) \neq \text{XOR}(0, 1)$). Distributed minimality is weaker than strong distributed minimality. We already showed previously that the OR function is distributed minimal. However it is not strong distributed minimal since input events $(0, 1)$ and $(1, 1)$ differ at exactly one position and $\text{OR}(0, 1) = \text{OR}(1, 1)$.

Definition 12 (Strong distributed non-minimality of program $\mathcal{DF}$). *Program $\mathcal{DF} : I \to O$ is* strong distributed non-minimal *for $I' \subseteq I$ iff there exists two input events $(i_1, \cdots, i_n)$ and $(i'_1, \cdots, i'_n)$ belonging to I' that differ exactly in one element, and the output that the program $\mathcal{DF}$ produces for $(i'_1, \cdots, i'_n)$ is equal to the output that it produces for $(i_1, \cdots, i_n)$. Formally,*

$$\begin{aligned}&\exists (i_1, \cdots, i_n), (i'_1, \cdots, i'_n) \in I' :\\ &\quad (\exists j \in [1, n] : i_j \neq i'_j \wedge \forall k \in [1, n] : k \neq j \implies i_k = i'_k) \wedge\\ &\qquad \mathcal{DF}((i_1, \cdots, i_n)) = \mathcal{DP}((i'_1, \cdots, i'_n))\end{aligned}$$

We now introduce strong distributed minimality property denoted as φ_{sdm} based on the definition of distributed minimality (Definition 11).

Definition 13 (Strong distributed minimality property φ_{sdm}). Strong distributed minimality property $\varphi_{sdm} \subseteq \Sigma^*$, *where* $\Sigma = I \times O$ *and* $I = I_1 \times \cdots \times I_n$ *is the set of all words belonging to* Σ^*, *such that for any word* $\sigma \in \varphi_{sdm}$, *for any two input-output events at different indexes in* σ, *let the inputs corresponding to the two event be* $(i_1, \cdots, i_n)$ *and* $(i'_1, \cdots, i'_n)$. *If only one input source value differ in* $(i_1, \cdots, i_n)$ *and* $(i'_1, \cdots, i'_n)$, *then the projection on outputs of the two input-output events should differ. Formally,*

$$\begin{array}{l}\forall \sigma \in \varphi_{sdm}, \\ \quad \forall i \in [1, |\sigma|], \forall j \neq i \in [1, |\sigma|], \\ \qquad \textit{let } \Pi_1(\sigma_i) = (i_1, \cdots, i_n), \Pi_2(\sigma_j) = (i'_1, \cdots, i'_n). \\ \qquad\quad (\exists x \in [1, n] : i_x \neq i'_x \wedge \forall y \in [1, n] : y \neq x \implies i_y = i'_y) \implies \Pi_2(\sigma_i) \neq \Pi_2(\sigma_j).\end{array}$$

Remark 12. Note that property φ_{sdm} is prefix-closed.

Example 11. Consider program $\mathcal{DP}$ to be the example program illustrated in Figure 4. Let $\sigma_1 = ((5000, 45), \mathsf{true}) \cdot ((11000, 45), \mathsf{false})$ be a prefix of an execution of this program which belongs to $\mathcal{L}(\mathcal{DP})$. We have $\sigma_1 \in \varphi_{dm}$. Consider another prefix of an execution of this program $\sigma_2 = ((5000, 45), \mathsf{true}) \cdot ((11000, 45), \mathsf{false}) \cdot ((12000, 45), \mathsf{false})$ where $\sigma_2 \in \mathcal{L}(\mathcal{P})$. Note that $\sigma_2 \notin \varphi_{sdm}$ since if we consider input-output events at index 2 and index 3, the projection of inputs in these events are resp. $(11000, 45)$ and $(12000, 45)$, and only salary information in these two input events differ. The output values of these two events are equal (true in both the events at index 2 and 3).

We now define strong distributed non-minimality property, which is negation of the distributed minimality property φ_{sdm} introduced in Definition 13.

Definition 14 (Strong distributed non-minimality property $\overline{\varphi_{sdm}}$). *Given alphabet* $\Sigma = I \times O$, *where* $I = I_1 \times \cdots \times I_n$, *property* $\overline{\varphi}_{sdm} \subseteq \Sigma^*$, *is the set of all words in* Σ^* *satisfying the following constraint:*

$$\begin{array}{l}\forall \sigma \in \overline{\varphi_{sdm}} : \\ \quad \exists i \in [1, |\sigma|], \exists j \neq i \in [1, |\sigma|], \ \textit{with } \Pi_1(\sigma_i) = (i_1, \cdots, i_n), \Pi_2(\sigma_j) = (i'_1, \cdots, i'_n) \ \textit{s.t.} \\ \qquad ((\exists x \in [1, n] : i_x \neq i'_x \wedge \forall y \in [1, n] : y \neq x \implies i_y = i'_y) \wedge \\ \qquad\quad (\Pi_2(\sigma_i) = \Pi_2(\sigma_j))\end{array}$$

Remark 13. Note, that property $\overline{\varphi}_{sdm}$ is extension closed, i.e., for any word σ that belongs to φ_{sdm}, every possible extension of σ also belongs to $\overline{\varphi}_{sdm}$. Formally, $\forall \sigma \in \Sigma^* : \sigma \in \overline{\varphi}_{sdm} \implies (\forall \sigma' \in \Sigma^* : \sigma \preccurlyeq \sigma' \implies \sigma' \in \overline{\varphi}_{sdm})$.

Example 12. Let us consider the example program $\mathcal{DP}$ illustrated in Figure 4. Consider a prefix of an execution of this program $\sigma_1 = ((5000, 45), \mathsf{true}) \cdot ((11000, 45), \mathsf{false}) \cdot ((12000, 45), \mathsf{false})$ where $\sigma_2 \in \mathcal{L}(\mathcal{P})$. Note that $\sigma_1 \in \overline{\varphi}_{sdm}$ since if we consider input-output events at index 2 and index 3, the projection of inputs in these events are resp. $(11000, 45)$ and $(12000, 45)$, and only salary information in these two input events differ. The output values of these two events are equal (true in both the events at index 2 and 3). Note that any extension of σ_1 also belongs to $\overline{\varphi}_{sdm}$.

Remark 14. Note that $\overline{\varphi_{sdm}}$ is the negation of property φ_{sdm}, where $\overline{\varphi_{sdm}} = \Sigma^* \setminus \varphi_{sdm}$. A word $\sigma \in \Sigma^*$ satisfies $\overline{\varphi_{sdm}}$ if $\sigma \in \overline{\varphi_{sdm}}$. It follows:

- $\forall \sigma \in \Sigma^*, \sigma \in \varphi_{sdm} \implies \sigma \notin \overline{\varphi}_{sdm}$;
- $\forall \sigma \in \Sigma^*, \sigma \in \overline{\varphi}_{sdm} \implies \sigma \notin \varphi_{sdm}$.

Theorem 2. *Given* $\mathcal{DF} : I \to O$ *where* $I = I_1 \times \cdots \times I_n$, *let* $\mathcal{L}(\mathcal{DP}) \subseteq \Sigma^*$ *with* $\Sigma = I \times O$, *where* $\mathcal{DP}$ *corresponds to the program for* $\mathcal{DF}$ *(*$\mathcal{DP}$ *is repeated execution of program* $\mathcal{DF}$*). The following properties hold:*

- $\mathcal{DF}$ *is strong distributed non-minimal iff* $\exists \sigma \in \Sigma^* : \sigma \in \mathcal{L}(\mathcal{DP}) \wedge \sigma \in \overline{\varphi}_{dm}$.
- $\mathcal{DF}$ *is strong distributed minimal iff* $\forall \sigma \in \Sigma^* : \sigma \in \mathcal{L}(\mathcal{DP}) \implies \sigma \in \varphi_{dm}$.

Theorem 3 (Minimality $\implies$ strong distributed minimality $\implies$ distributed minimality). *If program* $\mathcal{DF} : I \to O$ *is strong distributed minimal, for* $I' \subseteq I$, *according to Definition 11, then* $\mathcal{DF}$ *is also distributed minimal for* I' *as per Definition 9.*

If program $\mathcal{DF} : I \to O$ *is minimal, for* $I' \subseteq I$, *according to Definition 4, then* $\mathcal{DF}$ *is also strong distributed minimal for* I' *as per Definition 11, and is thus also distributed minimal for* I'.

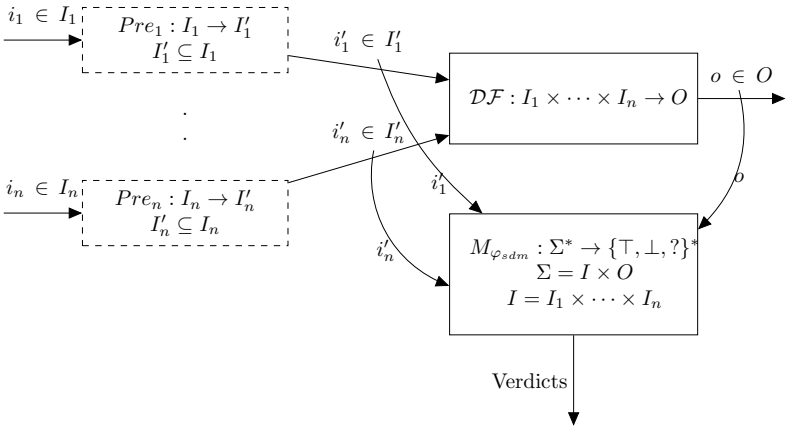


Fig. 11: Monitor M_{dm} in the distributed case.

4.2 Monitoring mechanisms to detect strong distributed (non) minimality

Similar to the monolithic case, we are interested in checking whether the inputs provided to an (untrusted) program $\mathcal{DF} : I_1 \times \cdots \times I_n \to O$ are minimized in the best possible way. We consider monitoring input-output behavior of program $\mathcal{DP}$ where $\mathcal{DP}$ is repeated execution of program $\mathcal{DF}$.

By monitoring $\mathcal{DP}$ (input-output behavior of several executions of program $\mathcal{DF}$), we are interested in checking whether an execution of $\mathcal{DP}$ satisfies strong distributed (non) minimality property.

The framework of the proposed monitoring approach in the distributed case, where the program has multiple input sources is depicted in Figure 11. We consider that there are multiple input sources $[1, n]$, and the set of all possible values for each input source $id \in [1, n]$ is denoted using I_{id}. Program $\mathcal{DP}$ requires an input from all its input sources to produce an output. An input event i that the user (or *testInputGenerator*) provides belongs to the set I, ($i \in I = I_1 \times \cdots \times I_n$). We assume that inputs from the user are first pre-processed by a data pre-processor (Pre), and the pre-processed input that belongs to the set I' (where $I' \subseteq I$) is fed as input to the *untrusted* program $\mathcal{DF}$. For each execution of $\mathcal{DF}$, the monitor observes both the pre-processed input and the output of $\mathcal{DF}$.

We consider that the monitor cannot observe and is not aware of the actual inputs $i_1, \cdots, i_n$ that the user provides. Moreover, pre-processors may or may not exist. We also assume that the monitor is unaware whether pre-processors exist or not, and that it does not know about their behavior. What the monitor observers at runtime is the pre-processed input i' that belongs to I' which also belong to I.

After each execution of program $\mathcal{DF} : I_1 \times \cdots \times I_n \to O$ (i.e., in every iteration of program $\mathcal{DP}$), the monitor observes the (pre-processed) input and the output of program $\mathcal{DF}$ in that particular iteration (step) of $\mathcal{DP}$.

For any word $\sigma \in \Sigma^*$ (current observation of execution of $\mathcal{DP}$) of length greater than 1, where $\Sigma = I \times O$ with $I_1 \times \cdots \times I_n$, $M_{\varphi_{sdm}}$ as per Definition 1 is a monitor for property φ_{sdm}. The monitor returns true ($\top$) when σ followed by any extension of it satisfies the distributed minimality property φ_{sdm}. The monitor returns $false$ ($\bot$) when the current observation of execution of $\mathcal{DP}$ followed by any extension of it violates φ_{sdm} (resp. satisfies $\overline{\varphi_{sdm}}$). It returns ? (unknown) for the current observation if the other two cases do not hold.

Similar to the monolithic case, checking whether the distributed minimality property is satisfied (resp. violated) for every extension of the current observation, needs to be reduced to checking whether the current observation satisfies (resp. violates) some constraints. We now introduce the function $\mathsf{sat}_{\varphi_{\mathsf{sdm}}}$ that is defined based on definitions of properties φ_{sdm} (resp. $\overline{\varphi_{sdm}}$). $\mathsf{sat}_{\varphi_{\mathsf{sdm}}}$ is used to check whether the current observation σ satisfies property φ_{sdm} (resp. $\overline{\varphi_{sdm}}$) .

σ	$M(\sigma)$
$((5000, 45), \mathsf{true})$	?
$((5000, 45), \mathsf{true}) \cdot ((11000, 51), \mathsf{false})$	?
$((5000, 45), \mathsf{true}) \cdot ((11000, 51), \mathsf{false}) \cdot ((4000, 21), \mathsf{true})$	?
$((5000, 45), \mathsf{true}) \cdot \mathbf{((11000, 51), false)} \cdot ((4000, 21), \mathsf{true}) \cdot \mathbf{((11000, 55), false)}$	$\bot$

Table 2: Example illustrating behavior of monitor M_{dm}.

The function $\mathsf{sat}_{\varphi_{\mathsf{sdm}}} : \Sigma^* \to \mathbb{B}$, takes an input-output word $\sigma \in \Sigma^*$ and it returns a Boolean as output. It is defined as follows:

$$\mathsf{sat}_{\varphi_{\mathsf{sdm}}}(\sigma) = \begin{cases} \mathsf{true} & \text{if } \begin{array}{l} \forall i \in [1, |\sigma|], \forall j \neq i \in [1, |\sigma|], \\ \quad \text{let } \Pi_1(\sigma_i) = (i_1, \cdots, i_n) \wedge \Pi_1(\sigma_j) = (i'_1, \cdots, i'_n). \\ \quad (\exists x \in [1, n] : i_x \neq i'_x \wedge \forall y \in [1, n] : y \neq x \implies i_y = i'_y) \\ \qquad \implies \Pi_2(\sigma_i) \neq \Pi_2(\sigma_j). \end{array} \\ \mathsf{false} & \textit{Otherwise} \end{cases}$$

$\mathsf{sat}_{\varphi_{\mathsf{sdm}}}$ checks whether a given word σ belongs to property φ_{sdm}. For any given word σ, $\mathsf{sat}_{\varphi_{\mathsf{sdm}}}(\sigma)$ is true if $\sigma \in \varphi_{dm}$, and is false otherwise, i.e., if it returns false, then $\sigma \in \overline{\varphi_{sdm}}$.

Similar to the monolithic case, the condition for the second case of the monitor for property φ_{sdm} can be simplified.

Proposition 5. *Given any word $\sigma \in \Sigma^*$, where $|\sigma| > 1$, we have that if $\sigma \in \overline{\varphi}_{sdm}$ then $(\forall \sigma' : \sigma \preccurlyeq \sigma', \sigma' \in \overline{\varphi_{sdm}})$.*

Remark 15. Similar to the monolithic case, regarding the condition of the $\top$ case (satisfaction of φ_{sdm}), checking whether the current observed word σ belongs to φ_{sdm} (i.e.,whether $\mathsf{sat}_{\varphi_{\mathsf{sdm}}}(\sigma)$ is true) is not sufficient, as it does not ensure that every extension of σ will also belong to φ_{sdm}. Thus, testing condition of the first case is not possible in general.

However, by providing additional knowledge to the monitor about all the input domains, and when they are bounded, checking the condition of the first case reduces to testing whether the current observation satisfies φ_{sdm}, and if it also covers all possible inputs.

Proposition 6. *Given any word $\sigma \in \Sigma^*$, where $\Sigma = I \times O$, with $I = I_1 \times \cdots I_n$, and when $|\sigma| > 1$, we have that if $(\text{in-ex}(I, \sigma) \wedge \mathsf{sat}_{\varphi_{\mathsf{sdm}}}(\sigma))$ then $(\forall \sigma' : \sigma \preccurlyeq \sigma', \sigma' \in \varphi_{sdm})$.*

Using Propositions 5 and 6, the conditions of the first two cases in $M_{\varphi_{sdm}}$ can be simplified. We present the simplified definition below, where the conditions of the first two cases are reduced to checking whether the observed input word satisfies some constraints.

Consider the property $\varphi_{sdm} \subseteq \Sigma^*$, where $\Sigma = I \times O$, and $I = I_1 \times \cdots \times I_n$. Let $\sigma \in \Sigma^*$ denote a finite input-output word over the alphabet $\Sigma = I \times O$ (current observation of an execution of $\mathcal{DP}$ which belongs to $\mathcal{L}(\mathcal{DP})$). The monitor for strong distributed minimality is denoted as $M_{\varphi_{sdm}}$. For $\sigma = \epsilon$ and any word σ of length 1, $M_{\varphi_{sdm}}(\sigma) =?$. $M_{\varphi_{sdm}}$ is defined as follows:

Definition 15 (Monitor for strong distributed minimality). *A monitor for property φ_{sdm} is a function $M_{\varphi_{sdm}} : \Sigma^* \to \mathcal{D}$, where $D = \{\top, \bot, ?\}$ is defined as follows:*

$$M_{\varphi_{sdm}}(\sigma) = \begin{cases} \top & \textit{if } |\sigma > 1| \wedge \text{in-ex}(I, \sigma) \wedge \mathsf{sat}_{\varphi_{\mathsf{sdm}}}(\sigma) \\ \bot & \textit{if } |\sigma > 1| \wedge \neg \mathsf{sat}_{\varphi_{\mathsf{sdm}}}(\sigma) \\ ? & \textit{Otherwise} \end{cases}$$

Example 13. Let us consider the example program $\mathcal{P}$ illustrated in Figure 4b. In Table 2, we present some example observations of an execution of program $\mathcal{DP}$ being monitored denoted as σ, and the verdict provided by the monitor for σ.

Proposition 7. *$M_{\varphi_{sdm}}$ in Definition 15 is a monitor for property φ_{sdm} as per Definition 1.*

5 Pre-deployment testing and minimiser synthesis via monitoring

The discussion and results of this section applies to both the monolithic and distributed cases. To simplify the presentation, we illustrate and discuss the results considering monitoring of monolithic minimality.

Let us consider the definition of monitor M_{φ_m} (Definition 8). In order to provide conclusive verdict $\top$ from an observed input-output word $\sigma \in \Sigma^*$, in addition to testing $\mathsf{sat}_{\varphi_m}(\sigma)$, the monitor has to be provided with information about the set of all possible inputs I, and we need to check whether every possible input appear in σ at least once (i.e., test whether in-ex(I, σ) holds).

In runtime monitoring, the word $\sigma \in \Sigma^*$ (observation of current execution of $\mathcal{P}$) that is fed to a monitor is of finite bounded length (σ and its length are both known). Thus, for any given $\sigma \in \Sigma^*$ and any set of inputs I, testing in-ex(I, σ) is straightforward, as illustrated in Algorithm 1.

Algorithm 1 in-ex(I, σ)

```
1: I' ← {}
2: for i ∈ [|1, |σ|] do
3:    inp ← Π1(σ_i)
4:    I' ← I' ∪ {inp}
5:    if |I'| = |I| then
6:       return true
7:    end if
8: end for
9: return false
```

Algorithm 1 (in-ex) requires the set of possible inputs I, and an input-output word $\sigma \in \Sigma^*$ (where $\Sigma = I \times O$) as input parameters. I' which is initially empty is used to keep track of the set of inputs seen in σ. While processing the sequence σ event by event to build I', if $|I'| = |I|$, then the algorithm returns true and terminates. In the worst-case, the for-loop runs for $|\sigma|$ times. After processing all the events in σ, if $|I'| \neq |I|$ then the algorithm returns false.

Note that before entering the for-loop in Algorithm 1 it is checked whether $|\sigma| < |I|$. If so, we can immediately return false.

Proposition 8 ($|\sigma| < |I|$). *When the length of the input-output word $\sigma \in \Sigma^*$ is less that the cardinality of the set of inputs, then* in-ex(I, σ) *is* false*:*

$$\forall \sigma \in \Sigma^* : |\sigma| < |I| \implies \text{in-ex}(I, \sigma) = \mathsf{false}$$

Remark 16 (Conclusive verdict $\top$ during runtime monitoring). In general, when performing online monitoring of program $\mathcal{P}$ (where $\sigma \in \Sigma^*$ is the current observation of execution of $\mathcal{P}$), and providing knowledge of the set of all possible inputs of $\mathcal{P}$ to the monitor, it is highly unlikely that σ covers all the inputs in I. Thus, as we can imagine, during runtime monitoring in-ex(I, σ) most likely will return false, and thus the condition of the first case (that provides conclusive verdict $\top$) in Definition 8 most likely does not hold, and thus we notice only verdicts ? or $\bot$ in practice.

However, monitor M_{φ_m} can be used for testing for minimality prior to deployment. In this case, the input observation fed to the monitor can be generated in such a way that it covers all the inputs in I (when I is finite and bounded).

5.1 Testing in a controlled environment via monitoring

When the set of inputs I is bounded, and when testing $\mathcal{F}$ in a controlled environment (i.e., when we have control over the inputs that are fed to $\mathcal{F}$), it is indeed possible to obtain a conclusive verdict (either $\top$ or $\bot$), upon observing a sequence σ of length $|I|$. We discuss this further via the monolithic case, which straightforwardly extends to the distributed case.

Algorithm 2 (DataMinTester) is for testing $\mathcal{F}$ via monitoring. In Algorithm 2, I' contains inputs that are not yet fed to the program $\mathcal{F}$. Initially, I' is assigned with the set of inputs I. In every iteration of

Algorithm 2 DataMinTester

1: $I' \leftarrow I$, $\sigma \leftarrow \epsilon$, $v \leftarrow ?$
2: **while** $(|I'| > 0 \wedge v == ?)$ **do**
3: $\quad i \leftarrow pickInp(I')$
4: $\quad o \leftarrow \mathcal{F}(i)$
5: $\quad \sigma \leftarrow \sigma \cdot (i, o)$
6: $\quad v \leftarrow M_{\varphi_m}(\sigma)$
7: $\quad I' \leftarrow I' \setminus \{i\}$
8: **end while**

```
inputPre(salary)
  if (salary < 10000)
    salaryRep = 1000
  else:
    salaryRep= 10000
return(salaryRep)
```

Fig. 12: Example of an input pre-processor (which is also a minimiser) for program $\mathcal{F}$ illustrated in Figure3a.

the while loop, an input i from the set I' is picked non-deterministically, which is fed to $\mathcal{F}$, and $\mathcal{F}(i)$ is assigned to o. The input-output event (i, o) is then fed to the monitor. Before proceeding to the next iteration, input i which is already considered in the current iteration is removed from the set I'.

Algorithm 2 (DataMinTester) can be considered as program $\mathcal{P}$ where program $\mathcal{F}$ is executed repeatedly. However, here, in every iteration we invoke $\mathcal{F}$ with a new input from the set I (i.e., input that has not been considered in the previous iterations). The while loop thus terminates after $|I|$ iterations.

After $|I|$ iterations of the algorithm, the input-output word σ that the monitor receives will be of length $|I|$, and in-ex(I, σ) will evaluate to true. The monitor M_{φ_m} certainly returns a conclusive verdict upon receiving an input-output word of length $|I|$.

Proposition 9. *Let $\sigma \in \Sigma^*$ be an execution of the* DataMinTester *(Algorithm 2), which is a sequence of input-output word fed to the monitor. The length of σ will be at most $|I|$, and the monitor will certainly return a conclusive verdict $\top$ or $\bot$ for σ of length $|I|$.*

Regarding conclusive verdict $\top$ (i.e, about satisfaction of minimality), the monitor cannot provide this verdict before observing a word of length $|I|$. Regarding conclusive verdict $\bot$ (i.e, about violation of minimality), the monitor may be able to provide this verdict before observing word of length $|I|$. In this case, the execution of the tester program can stop earlier soon after the monitor observes the sequence that satisfies non-minimality property.

5.2 Minimiser synthesis

In this section, by considering the monolithic case, we briefly present about possibility of synthesizing a pre-processor for $\mathcal{F}$ such that the composition of $\mathcal{F}$ with the synthesized pre-processor satisfies the data minimality principle.

An input pre-processor that does not change the behavior of $\mathcal{F}$, and makes $\mathcal{F}$ minimal when composed with it is called as a minimiser, defined as follows:

Definition 16 (Minimizer). *A pre-processor $Pre : I \rightarrow I$ is a monolithic* minimiser *for $\mathcal{F}$ iff Pre is a pre-processor for $\mathcal{F}$ and $\mathcal{F}$ is monolithic-minimal for* range(Pre).

Example 14. The input pre-processor presented in Figure 6 is not a minimiser for program $\mathcal{F}$ illustrated in Figure3a, since we can define a pre-processor with two cases. The input pre-processor presented in Figure 12 is also a minimiser for $\mathcal{F}$ in Figure3a.

We show that when the input domain I is bounded and known, by monitoring input-output behavior of $\mathcal{F}$, in addition to checking whether $\mathcal{F}$ is minimal (resp. non-minimal), it is also possible to synthesize a minimiser.

Algorithm for obtaining a minimiser. Algorithm 2 can be adapted for building a partitioning of the input domain I, where for every partition, program $\mathcal{F}$ produces the same output for any input belonging to that partition. The condition of the while-loop should now be $|I'| > 0$ since we need to continue the execution (irrespective of whether the monitor provides a conclusive verdict), to cover all the inputs to build a partitioning of the input domain. In each iteration of the while-loop:

- Let i be the input picked from the set of un-covered inputs, and let o be the output produced by $\mathcal{F}$.
- If an input partition corresponding to o already exists, then i is added to that input partition. Otherwise, a new partition corresponding to output o is created before proceeding to the next iteration.
- The while-loop terminates after $|I|$ iterations, and we have a partitioning of the input domain.

For each input partition, an element is chosen (non-deterministically) as input representative for that partition. $I' \subseteq I$ is the set of input representatives, and the algorithm returns a mapping from I to I', where for each input partition, every element belonging to that partition is mapped to its corresponding input representative.

Proposition 10. *Consider any program $\mathcal{F} : I \to O$, When I is known and $|I|$ is bounded, the algorithm for obtaining a minimiser discussed above terminates and it returns a minimizer for program $\mathcal{F}$.*

6 Implementation

The runtime monitoring mechanisms for checking minimality (resp. non-minimality) for both the monolithic and distributed cases have been implemented in Python. The main goal of this prototype implementation is to validate the feasibility and practicality of the proposed approaches (i.e., monitoring for (non) minimality at runtime, pre-deployment testing and synthesis of a minimiser).

Implementation of monitors. Regarding the implementation of monitors (e.g., the implementation of M_{φ_m}), implementation of functions sat_{φ_m} (resp. $\mathsf{sat}_{\overline{\varphi}_m}$) that checks whether a given trace (current observation of an execution of $\mathcal{P}$) is minimal (resp. non-minimal) is straightforward from their definitions. For the first case ($\top$) in the definition of M_{φ_m}, we also additionally need to check whether the current observation σ covers all the inputs (i.e., whether $inpExh(I, \sigma)$ holds, where I is the set of all possible inputs).

Monitoring for (non) minimality at runtime. For testing the usage of monitors at runtime for detecting (non) minimality, we wrapped the program (to be monitored) with a user simulator (test-input generator). The user simulator is an (infinite) while-loop, where in each iteration, the program (being monitored) is invoked with some input i chosen non-deterministically from the set of allowed inputs I, and the monitor is fed with the input i and the output o that the program returns. The loop terminates when the monitor returns a conclusive verdict ($\top$ or $\bot$). When the monitor returns conclusive verdict $\bot$, it also returns an evidence that shows violation of data minimality.

For example, when the program for computing benefits illustrated in Figure 3a composed with the input pre-processor illustrated in Figure 6 is considered as the program to be monitored, as expected the approach terminated and returned conclusive verdict $\bot$ with an evidence.

When the program for computing benefits in Figure 3a composed with the minimiser in Figure 12 is considered as the program to be monitored, we can notice that the user simulator does not terminate. As expected, coverage of all inputs is highly impossible in this approach and the monitor always returns verdict unknown (?).

Pre-deployment testing. We discussed about using the monitor to test the program in a controlled environment (Algorithm 2), and check whether it satisfies the data minimality principle. Algorithm 2 also has been implemented and tested.

When the program for computing benefits in Figure 3a composed with the minimiser in Figure 12 is considered as the program to be tested, as expected, the approach returned conclusive verdict $\top$ (i.e., the composition of the program in Figure 3a with the minimiser in Figure 12 satisfies data minimality).

```
################################################
## input numFlights: integer between 0 to 100#
################################################
def computeStatusLevel(numFlights):
    status = 0
    i = 0
    if numFlights<10:
        status = 0
    elif numFlights<20:
        status = numFlights -10
    elif numFlights<30:
        while i<= numFlights-20:
            status = status+numFlights
            i=i+1
        if status>150:
            status = 150
    else:
        status = 500
    return status
################################################
```

Fig. 13: Program $\mathcal{F}$ to compute a loyalty status.

Minimiser synthesis. The algorithm for synthesizing a minimiser discussed briefly in Section 5.2 has been also implemented for the monolithic case. When the program for computing benefits in Figure 3a is considered with $I = \{1, \cdots, 30000\}$, the minimiser synthesizer returned a minimiser where partitioning of I consists of two partitions. The set $\{1, \cdots, 9999\}$ is one partition and all the elements in this set are mapped to a representative chosen from it (e.g., 9999), and the set $\{10000, \cdots, 30000\}$ is the other partition, and all the elements from this set are mapped to an input representative chosen from it (e.g., 30000).

Example 15. Let us consider an airport facility that must provide services to customers depending on their status. The status level of a customer is determined depending on the number of flights taken by the customer in the previous year with its favorite company, *PrivaFly*. This number of flights information is disclosed by the airline company to the airport.

However, *PrivaFly* wants to adopt the best practices in personal data protection, and requires only the needed data to be disclosed. The airport services have their own policy to compute the status level, program $\mathcal{F}$ shown in Figure 13. The program `compStatusLevel` takes information about the number fo `flights`, and it returns a `status` level. If the number of flights is lower than 9, the status is 0. From 10 to 19 flights, the status level is `numFlights`-10. If the number of flights ranges from 20 to 29, the computation status involves a loop. Finally, over 30 flights, the status level is capped to 500.

Intuitively, there is no need to give a precise value for a number of flights between 0 and 9 and over 30. On the other hand, the exact number should be disclosed between 10 and 19.

The minimiser synthesis approach partitions the set of possible inputs (integer between 1 to 100) into 17 partitions, i.e., the set of possible outputs of the minimiser consists of 17 elements. Elements in $[0, 10]$ are grouped into a partition, and every element in this partition is mapped to a representative chosen from this partition. For elements between 11 to 24, there will be 14 partitions each partition consisting of one element. Elements in $[25, 29]$ are grouped into one partition (due to sealing of the status to 150 for flights between 20 to 29, the program returns the same output for input value between 25 and 29). All the remaining elements in $[30, 100]$ are grouped into a partition. The approach returns a minimiser in less than 0.02 seconds.

Remark 17 (Minimiser synthesis approach in [1]). The example illustrated in Figure 13, is one of the examples provided with the implementation described in [1]. Note that the minimiser synthesis approach

in [1] requires the source code of the program. In our monitoring approach, the program can be a black-box, in the example considered above, we only need to know that program `compStatusLevel` requires an integer between 1 to 100 (number of flights taken), and it returns information about the status. The approach in [1] is also very complex, involving symbolic execution of the program and the use of SAT solver to obtain a partitioning of the input space. Moreover, when the program contains loops (e.g., `compStatusLevel` program in Figure 13) the approach in [1] cannot synthesize a minimiser in general. By adding loop-invariants (that helps the symbolic executor to handle loops), the approach may generate a pre-processor, and whether the generated pre-processor is a minimiser or not depends on the added loop-invariants.

7 Conclusion and Future Work

Data minimisation is a privacy enhancing principle, stating that personal data collected should be no more than necessary for the specific purpose consented by the user. The data minimisation process aims to minimise the input data such that only data that is necessary is given to the program.

In this paper, we consider the problem of runtime monitoring of deterministic programs to detect (non) minimality. We propose monitoring mechanisms where a monitor observes the inputs and the outputs of a program, to detect violation of data minimisation policies. We formally define runtime monitors to check (non-)minimality for both the monolithic and the distributed case. We show that checking minimality is non monitorable in general for any of the cases, and that non-minimality for the monolithic and a strong version of the distributed cases is monitorable in general, but not for the normal distributed case. We prove that under certain conditions we can monitor both minimality and its negation for all cases. We describe a procedure that gives a definite answer on whether the program is minimal or not by using runtime monitoring in a controlled (pre-deployment) test environment, and also obtain a minimiser for the program under test. We also provide a proof-of-concept implementation.

In the near future, we plan to generalise the monitoring results discussed in this paper for other security policies for deterministic programs. We also intend to study and formalise the concept of data minimisation for non-deterministic systems, and explore on the monitoring and minimiser synthesis problem for such systems.

References

1. Antignac, T., Sands, D., Schneider, G.: Data Minimisation: A Language-Based Approach. In: IFIP Information Security & Privacy Conference (IFIP SEC'17). IFIP Advances in Information and Communication Technology (AICT), vol. 502, pp. 442–456. Springer Science and Business Media (2017)
2. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. 20(4), 14:1–14:64 (Sep 2011), http://doi.acm.org/10.1145/2000799.2000800
3. Blech, J.O., Falcone, Y., Becker, K.: Towards certified runtime verification. In: Aoki, T., Taguchi, K. (eds.) Formal Methods and Software Engineering: 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings. pp. 494–509. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
4. Bonakdarpour, B., Finkbeiner, B.: Runtime Verification for HyperLTL, pp. 41–45. Springer International Publishing, Cham (2016)
5. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. 18(6), 1157–1210 (Sep 2010), http://dl.acm.org/citation.cfm?id=1891823.1891830
6. Cohen, E.: Information transmission in computational systems. SIGOPS Oper. Syst. Rev. 11(5), 133–139 (Nov 1977), http://doi.acm.org/10.1145/1067625.806556
7. Colombo, C., Pace, G.J., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In: Cofer, D.D., Fantechi, A. (eds.) Formal Methods for Industrial Critical Systems, 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, Revised Selected Papers. Lecture Notes in Computer Science, vol. 5596, pp. 135–149. Springer (2008), https://doi.org/10.1007/978-3-642-03240-0
8. European Parliament and Council: Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) (apr 2016)

9. Falcone, Y., Fernandez, J., Mounier, L.: Runtime verification of safety-progress properties. In: Bensalem, S., Peled, D.A. (eds.) Runtime Verification, 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009. Selected Papers. Lecture Notes in Computer Science, vol. 5779, pp. 40–59. Springer (2009), http://dx.doi.org/10.1007/978-3-642-04694-0
10. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for Model Checking HyperLTL and HyperCTL, pp. 30–48. Springer International Publishing, Cham (2015)
11. Havelund, K., Goldberg, A.: Verify your runs. In: Verified Software: Theories, Tools, Experiments: First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions. pp. 374–383. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
12. Leucker, M.: Runtime verification for linear-time temporal logic. In: Bowen, J.P., Liu, Z., Zhang, Z. (eds.) Engineering Trustworthy Software Systems - Second International School, SETSS 2016, Chongqing, China, March 28 - April 2, 2016, Tutorial Lectures. Lecture Notes in Computer Science, vol. 10215, pp. 151–194 (2016), https://doi.org/10.1007/978-3-319-56841-6
13. Leucker, M., Schallhart, C.: A brief account of runtime verification. Journal of Logic and Algebraic Programming 78(5), 293–303 (may/june 2009), http://dx.doi.org/10.1016/j.jlap.2008.08.004
14. Organisation for Economic Co-operation and Development: The OECD Privacy Framework. Guidelines, Organisation for Economic Co-operation and Development (2013), chapter 1. Recommendation of the Council concerning Guidelines governing the Protection of Privacy and Transborder Flows of Personal Data (2013)
15. Smith, G.: On the foundations of quantitative information flow. In: de Alfaro, L. (ed.) Foundations of Software Science and Computational Structures, Lecture Notes in Computer Science, vol. 5504, pp. 288–302. Springer Berlin Heidelberg (2009)
16. US Secretary's Advisory Committee on Automated Personal Data Systems: Records, Computers and the Rights of Citizens. Report DHEW NO. (OS)73-94, US Secretary's Advisory Committee on Automated Personal Data Systems, Brussels, Belgium (July 1973), chapter IV: Recommended Safeguards for Administrative Personal Data Systems